HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Simulation of two-dimensional heat conduction in Julia programming language using CUDA

By Eslam Abdelfatah E.Mohamed
Matriculation Nr.: 30740

Examiner: Prof. Dr.-Ing. Lothar Berger

Second supervisor: M. Sc. Stephan Scholz

A thesis submitted in partial fulfillment of the requirements for the degree
B. Eng. of Electrical Engineering and Information Technology

Faculty of Electrical Engineering and Computer Science
Ravensburg-Weingarten University of Applied Science
Germany
April 2022

Simulation of two-dimensional heat conduction in Julia programming language using CUDA

by Eslam Abdelfatah E.Mohamed

Abstract

Heat is vital when work and energy are involved in phase transitions, according to this fact, heat conduction qualities as well as the properties of molecules inside the bodies needed to be analyzed. Computation of heat transfer has been a considerable challenge during the last decade, especially when time is considered as a component with an unaffordable price.
The heat equation is a parabolic partial differential equation that represents how temperature varies in space over time. It may also be described as the process which reflects how heat is transferring from a higher temperature median to a lower temperature one. In some complex occasions it is nearly impossible to calculate the heat diffusion manually, that is why Julia programming language is utilized to solve one and two-dimensional heat equation problems by using CUDA, which is configured to accelerate the calculation through the graphics processing unit (GPU).

Acknowledgements

I would like to thank my supervisors, Prof. Dr.-Ing. Lothar Berger and Mr. Stephan Scholz, for their constant support, patience, and guidance throughout my bachelor's thesis. Over six months, I gained experience not only in the technical sector, but also in how to cope with difficulties, how to handle working under pressure by turning the negative vibe into a motivational one otherwise, I wouldn't be able to conquer and take any steps forward.
I can't express how grateful I am to my family for their love, understanding, support, and inspiration.
Finally, I'd like to thank my friends who have supported me during difficult times and have been a part of my adventure at Ravensburg-Weingarten University.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern graphics and image processing are handled relatively efficiently by computer graphics processing units (GPUs). Due to their highly parallel structure, GPUs are more effective than general-purpose CPUs for techniques that need parallel processing of huge blocks of data. Furthermore, modern GPUs can execute tens of thousands of threads at the same time.
A significant difference in computation operations was made by compute unified device architecture (CUDA). It rapidly runs the tasks and renders high-resolution images for all outcomes. CUDA is designed to be coded in programming languages such as C, C++, and Julia. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL, which required advanced skills in graphics programming.
DifferentialEquations.jl, DiffEqGPU.jl and CUDA.jl are packages developed in the Julia programming language and includes functionality for making use of GPUs in the differential equation solvers.

Figure 1.1: Graphic Card NVIDIA GeForce GTX 1050 Ti [1]

Figure 1.1 shows the Nvidia graphics card which we use for our data processing, It has seven hundred sixty eight CUDA cores, A memory speed of seven Gbps, stander memory configuration of four GB, memory interface GDDR5 , CUDA supporter and capability of simultaneous multi-projection [2].

1.1 Motivation

A graphics processing unit (GPU) is a specialized electronic circuit that can manipulate and alter memory quickly in order to speed up the production of images in a frame buffer for display. Nvidia popularized the term "GPU" in 1999 when it promoted the GeForce 256 as the world's first graphic processing unit. A single-chip processor with integrated transform, lighting, triangle setup/clipping and rendering engines are capable of processing at least 10 million polygons per second [3].
Originally, GPUs were only used to render visuals for video games. Many calculations are required to create a figure from recorded data. In comparison to the CPU, it takes far less time to create a vibrant, high-quality image.
CUDA programming language has high efficacy at solving complex mathematical operations by calling CuArray, which can operate arrays by passing them to GPU cores or using one of the methods defined for gpuArray objects to establish an array directly on the GPU. These kinds of arrays are specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.
The schematic figure 1.2 shows an example distribution of chip resources for the CPU versus GPU.

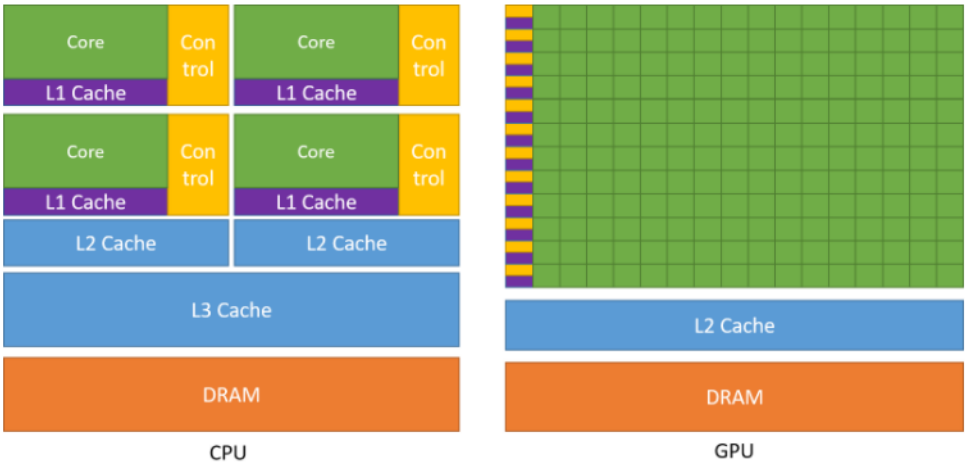
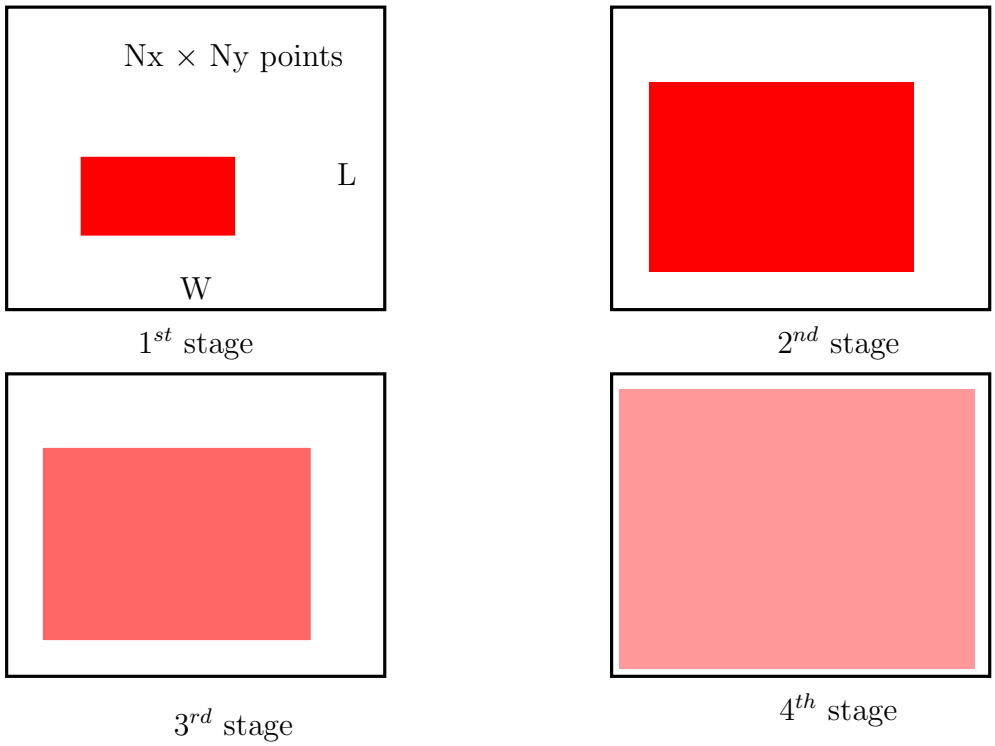


Figure 1.2: GPU Devotes More Transistors to Data Processing [4]

The simplest 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 makes them better suited to different tasks [5].
The 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 number of cores, all of these cores share the same device memory, which means input and output data from the GPU take 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, deals with multiple tasks simultaneously and performing relatively simple operations on them. However, if you're not doing thousands of "stuff" in parallel, it's not going to be fast [6].

1.2 Problem Statement

The study of heat conduction theory was first developed by Joseph Fourier in 1822 for the purpose of modeling how a quantity such as heat diffuses through a given region. The heat equation is a parabolic differential equation and it is considered basic to the broader subject of partial differential equations. The problem, is determining how to derive the next step value from the previous one.
In this thesis statement, we simulate one-dimensional and two-dimensional heat conduction by assuming a homogeneous rod in a one-dimensional case it has the same properties at every point, it is uniform without irregularities, with a length "L", then applying a heat source somewhere on the rod, By dissect the rod into M-nodes and visualizing the heat spreading along this nodes, we can notice how the temperature of the electrons in the atoms rise and keep rising by time passing, The heat spot is expanding till it reach to the boundaries which we are controlling once with Dirichlet's method "Fixed boundary condition" and once with Neumann's method "Heat flux boundary condition".
For two-dimensional case, a plate with a length of L and a width of W. By the same technique, set a heat source some were on the plate , the heat source has temperature 500 [K]. The plate shows the spreading of the heats spot. the plate is divided in the x-direction with Nx-steps and in y-direction with Ny-steps.



Basic idea of heat extension over the plate

The diagram shows the diffusion of the heat source with 500 kelvin temperature (1^{st} stage). The heat spreads in a 360-degree direction as in the 2nd stage. Heat goes through the direct microscopic exchange of kinetic energy of particles such as molecules in 3^{rd} stage. In 4^{th} stage the heat flows so that the body and the surroundings reach the same temperature, at which point they are in thermal equilibrium.

The temperature of each point is stored in a cell of an array. Each cell is a vector value that has a magnitude and direction. We will represent this array by using a CUDA array that provides a fully functional GPU array, which copy values simultaneously. and with same strategy it past all out outcome immediately.

Two-dimensional heat matrix

One of the main factors in heat conduction is the number of the cells, as when the number of the cells changes, the values of x-grid spacing and y-grid spacing change. The grid spacing is the distance between two cells.This affects the whole calculation process.
The simulation in time is implemented first with a manual Euler integration theory (full-discretization). A second will be through the DifferentialEquations.jl package, which implements the results as (semi-discretization).

Application Problem

In reality, we have a lot of applications that describe our problem. One of these applications is the micro heat pipe array. A heat pipe is a two-phase heat transfer device with a very high effective thermal conductivity that is made up of a series of micro heat pipe channels and is used to create a small heat dissipation device that can effectively remove heat from an electronic chip. Each of the array's channels functions as a separate heat transfer device.

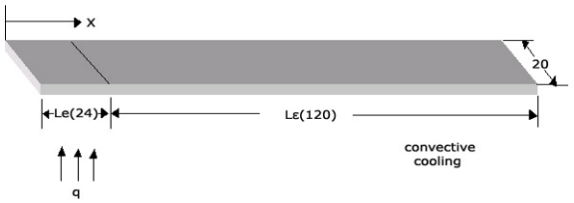


Figure 1.3: External view of the micro heat pipe [7]

The heat sinks have been presented previously by Sobhan et al. (2000). Originally micro heat pipes were developed as a microscale heat removal device for miniaturized applications, such as electronics cooling. The average length of micro heat pipes is a few centimeters and they have a hydraulic (internal) diameter between 50 and 600 μm.

1.3 Strategy of solution

The heat equation is a parabolic partial differential equation, so we will start by learning the characteristics of the partial differential equation that describes a value by how it changes in multiple directions, how it changes in the (x, y) physical directions, and how it changes in time.
The best way to solve a PDE problem is to convert it into an ODE problem, so we will go through spatial discretization using finite difference methods to convert the partial differential equation into an ordinary differential equation.
Typically, we have specified some boundary conditions, which specify the value of the solution or its derivatives along the boundary of a region, and some initial conditions, which specify the value of the solution or its derivatives for some initial time.
We will first set this boundary condition to the Dirichlet boundary condition, then to the Neumann boundary condition, and compare the results.
The input is defined as a matrix. Each matrix has its own (i, j) vector value. So we will use the dot product to multiply matrices together.
Discretization will be eliminated twice. First, as full-discretization by ruining the Euler's method ("a numerical method that can be used to approximate the solution to an initial value problem with a differential equation"), we will run it manually by using the mathematical Euler formula and a for loop to make it several times, which will give us the ability to see changes in the value of the heat matrix.
Secondly, with the semi-discretization method, which will be introduced under the DifferentialEquations.jl and DiffEqGPU.jl packages. These packages are inserted into the Julia language and have a different number of algorithmic methods that we will choose among them.
We aim to utilize the CUDA programming language to solve the heat equation. As a hardware tool, the GTX 1050 Ti is used, and as a software language, the Julia programming language, which is among the extensions of Atom, Visual Studio Code, and Visual Studio Codium. All of these are open-source text and source code editors, with which we got in touch during the period of the thesis.
We will run the computation twice: once on the motherboard by creating the system on the central processing unit and once on the graphic card by using the CuArray command to activate the graphic processing unit. Then see the differences in the results. A comparison of efficiency and time cost will be made.

Chapter 2

The Heat conduction

In section 2.1 of this chapter, we introduce the mathematical basics of partial differential equations. In section 2.2, we discuss the analytical and numerical heat conduction models. Finally, numerical integration methods are presented in section 2.3.

2.1 Partial differential equations

A partial differential equation, or simply a PDE, is a mathematical equation involving two or more independent variables, an unknown function depending on those variables, and partial derivatives of the unknown function with respect to those variables.

$$E(u) = \int_{\Omega} f(x, u, \nabla u, dx)) \tag{2.1}$$

where $\Omega \subset \mathcal{R}^n$. n is a domain, $x = (x_1, ..., x_n)$, $u = u(x)$ and $\nabla u = (u_{x1}, ...u_{xn})$ and $u_{x1} = \dfrac{\partial u}{\partial x_1}$ [8].
For linear PDEs in two dimensions there is a simple classification in terms of the general equation $Au_{xx} + Bu_{xy} + Cu_{yy} = f(x, y)$, where $u = u(x, y)$.
The A, B and C coefficients are real and in general can also be functions of x and y as
$u_{xx} = \dfrac{\partial^2 u}{\partial x^2}$ second partial derivative relative to i-th inputs,
$u_{xy} = \dfrac{\partial^2 u}{\partial x \partial y}$ second partial derivative relative to the i-th and j-th inputs,
$u_{yy} = \dfrac{\partial^2 u}{\partial y^2}$ second partial derivative relative to j-th inputs.

This classification concept is local as it is function of x and y. If $f(x, y)$ is zero everywhere, then the linear PDE is homogeneous otherwise, it is in-homogeneous. The PDEs of this type are classified by the value of discriminant "D_λ" of the eigenvalue problem, where $D_\lambda = B^2 - 4AC$. A simple classification is shown at table 2.1. Elliptic, parabolic, and hyperbolic partial differential equations of second order have been widely studied since the beginning of the twentieth century.

D_λ	**Type**	**Eigenvalues**
$D_\lambda < 0$	Elliptic	All positive or all negative.
$D_\lambda = 0$	Parabolic	All positive or all negative, except one that is zero.
$D_\lambda > 0$	Hyperbolic	Only one negative eigenvalue and all the rest are positive or vice versa.

Table 2.1: Coefficients of partial differential equation [9]

However, we focus on the parabolic PDE. By changing the independent variables, parabolic equations can be turned into a form of heat equation.

$$\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial x^2} \tag{2.2}$$

is a parabolic equation because it describes time-dependent and dissipative processes such as diffusion that are evolving toward a steady state. Each of these classes should be investigated separately as different methods are required for each. The next point to emphasize is that the method for numerically solving differential equations.

Finite Difference

The PDE is to approximate all the derivatives by finite differences. We divide the domain in space using a mesh $x_0, x_1, ..., x_N$ and in time using a mesh $t_0, t_1, ..., t_T$. First, we assume a uniform partition both in space and in time, so that the difference between two consecutive space points will be Δx and between two consecutive time points will be $\triangle t$ like

$$x_i = x_0 + i\Delta x, \qquad i = 0, 1, \cdots, \Delta x_N.$$

$$t_j = x_0 + j\Delta t, \qquad j = 0, 1, \cdots, \Delta t_T.$$

The Taylor series method,consider a Taylor expansion of an analytical function $u(x)$

$$u(x+\Delta x) = u(x) + \sum_{n=1}^{\infty} \frac{\Delta x^n}{n!} \frac{\partial^n u(x)}{\partial x^n} = u(x) + \Delta x \frac{\partial u}{\partial x} + \frac{\Delta x^2}{2!}\frac{\partial^2 u}{\partial x^2} + \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3} + \cdots, \tag{2.3}$$

then for the first derivative we obtain

$$\frac{\partial u}{\partial x} = \frac{u(x+\Delta x) - u(x)}{\Delta x} - \frac{\Delta x}{2!}\frac{\partial^2 u}{\partial x^2} - \frac{\Delta x}{3!}\frac{\partial^3 u}{\partial x^3} - \cdots. \tag{2.4}$$

If we break the right-hand side of the last equation after the first term, for $\Delta x << 1$. The equation becomes

$$\frac{\partial u}{\partial x} = \frac{u(x+\Delta x) - u(x)}{\Delta x} + O(\Delta x) = \frac{\Delta_i u}{\Delta x} + O(\Delta x), \tag{2.5}$$

where

$$\Delta_i u = u(x + \Delta x) - u(x) := u_{i+1} - u_i$$

is called a forward difference. The backward expansion of the function $f(u)$ and $\Delta x << 1$ can be written as

$$\frac{\partial u}{\partial x} = \frac{u(x) - u(x - \Delta x)}{\Delta x} + O(\Delta x) = \frac{\bigtriangledown_i u}{\Delta x} + O(\Delta x) \tag{2.6}$$

where

$$\bigtriangledown_i u = u(x) - u(x - \Delta x) := u_i - u_{i-1}.$$

We can see that both forward and backward differences are of the order $O(\Delta x)$. By combining these two approaches, we derive a central difference, which yields a more accurate approximation, if we subtract equation (2.5) from equation (2.6) we get

$$\frac{\partial u}{\partial x} = \frac{u(x + \Delta x) - u(x - \Delta x)}{2\Delta x} + O(\Delta x^2). \tag{2.7}$$

The second derivative can be found in the same way by using the linear combination of different Taylor expansions. For instance, consider

$$u(x + 2\Delta x) = u(x) + 2\Delta x \frac{\partial u}{\partial x} + \frac{(2\Delta x)^2}{2!}\frac{\partial^2 u}{\partial x^2} + \frac{(\Delta x)^3}{3!}\frac{\partial^3 u}{\partial x^3} + \cdots \tag{2.8}$$

subtracting from the last equation, equation (2.3) and multiplied by two we get

$$u(x + 2\Delta x) - 2u(x + \Delta x) = -u(x) + \Delta x^2 \frac{\partial^2 u}{\partial x^2} + \Delta x^3 \frac{\partial^3 u}{\partial x^3} \cdots. \tag{2.9}$$

Hence we can approximate the second derivative as

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x + 2\Delta x) - 2u(x + \Delta x) + u(x)}{\Delta x^2} + O(\Delta x). \tag{2.10}$$

Similarly one can obtain the expression for the second derivative in terms of backward expansion like

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x - 2\Delta x) - 2u(x - \Delta x) + u(x)}{\Delta x^2} + O(\Delta x). \tag{2.11}$$

Expression for the central second derivative reads

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x + \Delta x) - 2u(x) + u(x - \Delta x)}{\Delta x^2} + O(\Delta x^2). \tag{2.12}$$

This is a centred finite difference approximation with 2nd order accuracy. Thus, approximation of $u_t(x, t)$ could be $\frac{u_i^{n+1} - u_i^n}{\Delta t}$ with 1st order accuracy. Then the PDE in equation can be discretized as

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}. \tag{2.13}$$

The solution of PDEs involves arbitrary functions, Additional conditions are needed. These conditions can be given in the form of initial and boundary conditions i,e.,

$$\begin{cases} u_t = ku_{xx} & (x,t) \in (0,L) \times (0,T_f) \\ u(x,0) = 0 & IC \\ u(L,t) = 0 & BC \end{cases} \tag{2.14}$$

Initial conditions (IC) define the values of the dependent variables at the initial stage (e.g. at $t = 0$), whereas the boundary conditions (BC) give the information about the value of the dependent variable or its derivative on the boundary of the area of interest.

Boundary Condition

Boundary conditions are constraints that must be satisfied in order to solve a boundary value issue. A boundary value problem is a differential equation (or system of differential equations) that must be solved in a domain with a set of known conditions on the boundary [10].
In Figure 2.1 Ω is the function domain, $\partial\Omega$ is the boundary of the domain.

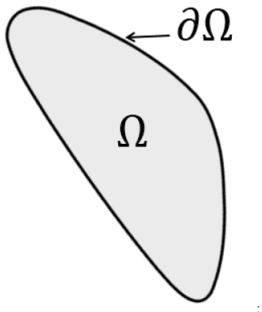


Figure 2.1: Resolution domain and boundary [10]

There are five types of boundary conditions: Dirichlet, Neumann, Robin, Mixed, and Cauchy, within which Dirichlet and Neumann are predominant and that what we focus on.
Dirichlet boundary conditions may also be referred to as a fixed boundary condition. In heat transfer problems, this condition corresponds to surface is held at a fixed temperature. As an example, on a plate with Dirichlet boundary conditions, there is heat transfer at the surface, while the surface remains at the temperature of the phase change process. $u(0,t) = v_1(t)$, $u(L,t) = v_2(t)$ where $v_1(t)$ and $v_2(t)$ remain at constant temperature.
Neumann boundary conditions represent the heat flux across the boundaries. When imposed on an ordinary or a partial differential equation, the condition specifies the values of the derivative applied at the boundary of the domain. In heat transfer problems, prescribed heat flux from a surface would serve as a boundary condition. If the flux is equal to zero, the boundary conditions describe the ideal heat insulator with heat diffusion $u_x(0,t) = w_1(t), u_x(L,t) = w_2(t)$ [11].

2.2 Heat Equation

The heat equation is a partial differential equation that describes how heat moves through a region over time.
Let $u(\overrightarrow{x}, t)$ represent the heat at a point $\overrightarrow{x}$ in n-dimensional space at time t, and let $u_0(\overrightarrow{x}) = u(\overrightarrow{x}, 0)$ be the initial distribution of heat.

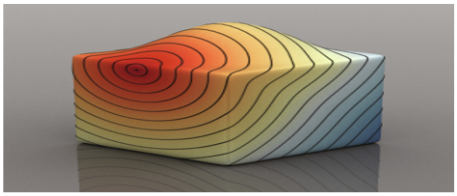


Figure 2.2: Heat kernel [12]

In the simplest case, heat distributes over time according to the homogeneous heat equation

$$\frac{\partial u(\overrightarrow{x}, t)}{\partial t} = \alpha \nabla^2 u(\overrightarrow{x}, t). \tag{2.15}$$

For any set of time the heat equation can be defined as

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \frac{\partial^2 u}{\partial x_3^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2} \right). \tag{2.16}$$

A solution to the homogeneous heat equation can be used to construct solutions to a more general in-homogeneous heat equation in which the right-hand in Equation 2.16 side includes an additional $f(x, t)$ term that models heat sources or sinks. Since the heat equation is defined with respect to continuous functions, but computers operate only on discrete values, we look for an approximate solution to the heat equation by discretizing space-time.

2.2.1 Discretization for one-dimensional case

Discretization is the method of converting continuous variables, models, or functions into discrete ones. This procedure is typically used as the initial step in preparing the heat equation for numerical evaluation and implementation on digital computers. From Equation 2.13 displays the temperature change at every discretized space xi in the one-dimensional body under consideration of the boundary conditions. The boundary condition for both ends of the rod are fixed (Dirichlet boundary condition) as $u_{-1} = 0$ and $u_{Nx+1} = 0$. As the both value u_{-1}, u_{Nx+1} are imaginary points Nx is the number of discretization points.

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{\alpha}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

By applying the Forward in Time Central in Space Scheme (FTCS) where place the time derivative by the forward differencing scheme and the space derivative by the central difference scheme.

$$u_i^{n+1} - u_i^n = \frac{\alpha \Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n) \tag{2.17}$$

since

$$u_i^{n+1} = (\frac{\alpha \Delta t}{\Delta x^2})u_{i+1}^n + (1 - 2\frac{\alpha}{\Delta x^2})u_i^n + (\frac{\alpha}{\Delta x^2})u_{i-1}^n \tag{2.18}$$

if we considered the $\theta = \dfrac{\alpha \Delta t}{\Delta x^2}$ The explicit nature of the difference method can then be expressed in matrix

$$\begin{pmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \\ \vdots \\ \\ u_{Nx-1}^{n+1} \end{pmatrix} = \begin{pmatrix} 1-2\theta & \theta & 0 & 0 & & & & \\ \theta & 1-2\theta & \theta & 0 & & & & \\ 0 & \theta & 1-2\theta & \theta & & & & \\ & \ddots & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & \ddots & & \\ & & & 0 & \theta & 1-2\theta & \theta & \\ & & & 0 & 0 & \theta & 1-2\theta & \end{pmatrix} \begin{pmatrix} u_1^n \\ u_2^n \\ u_3^n \\ \\ \vdots \\ \\ u_{Nx-1}^n \end{pmatrix}. \tag{2.19}$$

For Neumann boundary condition. A boundary condition on the derivative ux is given rather than a condition on the value of u itself. Now the boundary condition is

$$u_x(0,t) = w_1(t), \quad u_x(L,t) = w_2(t).$$

To approximate the Neumann Boundary Condition, we derive a second order approximation

$$\frac{u_1 - u_{-1}}{2\Delta x} = w_1(t), \quad \frac{u_{i+1} - u_{i-1}}{2\Delta x} = w_2(t) \tag{2.20}$$

u_{-1} and u_{Nx+1} are imaginary points, we know u_0^{n+1} is approximated by u_{-1}^n, u_0^n and u_1^n. Also, u_{Nx}^{n+1} approximated by u_{Nx-1}^n, u_{Nx}^n and u_{Nx+1}^n.

$$u_0^{n+1} = \theta u_{-1}^n + (1-2\theta)u_0^n + \theta n_1^n \tag{2.21}$$

and

$$u_{Nx}^{n+1} = \theta u_{Nx-1}^n + (1-2\theta)u_{Nx}^n + \theta U_{Nx+1}^n. \tag{2.22}$$

Now, use $u_{-1} = u_1 + 2w_1(t)\Delta x$ and $u_{Nx+1} = u_{Nx-1} + 2w_2(t)\Delta x$, According to equation 2.20.

$$u_0^{n+1} = \theta(u_{-1}^n - 2w_1(t)\Delta x) + (1-2\theta)u_0^n + \theta u_1^n \tag{2.23}$$

since

$$u_0^{n+1} = 2\theta u_1^n + (1-2\theta)u_0^n - 2\theta w_1(t)\Delta x \tag{2.24}$$

then

$$u_0^{n+1} + 2\theta w_1(t)\Delta x = 2\theta u_1^n + (1-2\theta)u_0^n. \tag{2.25}$$

Similarly, we have the u_{Nx} approximation

$$u_i^{n+1} + 2\theta w_2(t)\Delta x = 2\theta u_i^n + (1-2\theta)u_{i-1}^n \tag{2.26}$$

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

$$\begin{pmatrix} u_0^{n+1} \\ u_1^{n+1} \\ u_2^{n+1} \\ \\ \vdots \\ \\ u_{Nx}^{n+1} \end{pmatrix} = \begin{pmatrix} 1-2\theta & 1+\theta & 0 & 0 & & & & \\ \theta & 1-2\theta & \theta & 0 & & & & \\ 0 & \theta & 1-2\theta & \theta & & & & \\ & \ddots & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & \ddots\ddots & & \\ & & & 0 & \theta & 1-2\theta & \theta & \\ & & & 0 & 0 & 1-2\theta & 1+\theta \end{pmatrix} \begin{pmatrix} u_0^n + 2\theta w_1(t)\Delta x \\ u_1^n \\ u_2^n \\ \\ \vdots \\ \\ u_{Nx}^n + 2\theta w_2(t)\Delta x \end{pmatrix} \tag{2.27}$$

We have this because

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

For $i = 1, 2, ..., Nx-1$ we use the derived finite difference expression 2.21 and 2.22 for u_0 and u_{Nx}.

2.2.2 Two-dimensional heat equation

In two dimensions, we have $\overrightarrow{x} = (x, y)$ and $u(\overrightarrow{x}, t) = u(x, y, t)$, in which case this equation translates to

$$\frac{\partial u(x,y,t)}{\partial t} = \alpha\left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2}\right). \tag{2.28}$$

Suppose that in two dimensions we discretize the dimensions x, y, and t into points spaced Δx, Δy, and Δt apart, respectively. In the discretized space-time, each integer tuple $(i, j, n) \in Z^3$ corresponds to a point $(x, y, t)) = (i\Delta x, j\Delta y, n\Delta t)$ in continuous space.Because of the discretization, however, we can only compute a function $U_{i,j}^n$ which is an approximation to the solution, that is,

$$U_{i,j}^n \approx \alpha(u(i\Delta x, j\Delta y, n\Delta t)) \tag{2.29}$$

Because the heat equation incorporates continuous derivatives, it must also be discretized. Approximating derivatives with finite-difference methods is a standard technique. It turns out that one typical approximation for the first derivative with regard to time for the heat equation is

$$\frac{\partial u(x,y,t)}{\partial t} \approx \alpha\left(\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t}\right) \tag{2.30}$$

and for the spatial second derivatives, a popular approximation is

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} \approx \alpha\left(\frac{u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n}{\Delta x^2}\right). \tag{2.31}$$

```
function diffusion2d_x!(dx,x,Nx, Ny, dx)
for iy in 1 : Ny
    for ix in 2 : Nx-1
        i = (iy-1)*Nx + ix
        dx[i] =  (x[i-1] - 2*x[i] + x[i+1])/ dx^2
    end
end
```

A simple loop in Julia that performs the x-axis diffusion, which shown in equation 2.31.

$$\frac{\partial^2 u(x,y,t)}{\partial y^2} \approx \alpha(\frac{u^n_{i,j-1} - 2u^n_{i,j} + u^n_{i,j+1}}{\Delta y^2}). \tag{2.32}$$

```
function diffusion2d_y!(dx,x,Nx, Ny, dy)
for ix in 1 : Nx
    for  iy in 2 : Ny-1
            i = (iy-1)*Nx + ix
            dx[i] = dx[i] + (x[i-Nx] - 2*x[i] + x[i+Nx])/dy^2
    end
end
```

A simple loop in Julia that performs the computation in equation 2.32.
Substituting the approximations equation 2.31, equation 2.32 into equation 2.28. we get

$$\frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} = \alpha \left(\frac{u^n_{i-1,j} - 2u^n_{i,j} + u^n_{i+1,j}}{\Delta x^2} + \frac{u^n_{i,j-1} - 2u^n_{i,j} + u^n_{i,j+1}}{\Delta y^2} \right). \tag{2.33}$$

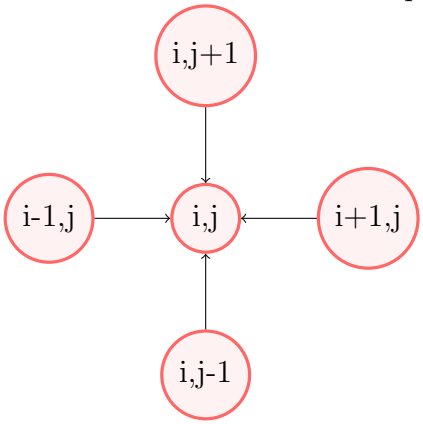
Although u(x,y,t) and $U^n_{i,j}$ are functions that are conceptually defined everywhere, we can only simulate a finite grid, that is, $0 \leq i \leq W$ and $0 \leq j \leq L$ for a finite time $0 \leq n \leq T_f$. The choice of the spatial steps Δx and Δy are determined by the required spatial resolution. With the spatial steps set, the time step is limited by the stability of the numerical approximation of the equation [13]

$$\partial t \leq \frac{\Delta x^2 \Delta y^2}{2\alpha(\Delta x^2 + \Delta y^2)}. \tag{2.34}$$

Stencil Computations

A stencil is a geometric arrangement of a nodal group that relates to the point of interest by using a numerical approximation routine. Stencils are the basis for many algorithms to numerically solve PDE. Two examples of stencils are the five-point stencil and the Crank–Nicolson method stencil. The term "stencil" was coined for such patterns to reflect the concept of laying out a stencil in the usual sense over a computational grid to reveal just the numbers needed at a particular step [14][15]. The approximation scheme in equation 2.33 belongs to the stencil computation category.
As we mentioned, to compute u at a point $(i, j, n + 1)$ in step $n + 1$, we require

five points from the previous time step n. A stencil is a set of points that must be accessed in order to calculate a value at the next time step.



Visualization of the nodes

In a stencil computation, the computation of the value at a point (i,j) in space at time step n requires only "local" values, that is, values from neighboring points of (i,j) from a few previous time steps. In general, one can use other approximation schemes for solving equation 2.28 besides equation 2.33. Many of these alternative schemes can also expressed as stencil computations. For example, one can use a higher-order approximation to derivatives, which would require a larger stencil.
A simple loop in Julia which we referred above shows a simple nested loop which performs the stencil computation in equation 2.33 goes once by once for each value of the heat matrix for time steps n ranging between t_0 and t_1. In this code, $u(x, y, t)$ conceptually stores the values for $u^n_{i,j}$ in some abstract array data structure.

```
function diffuse!(u, a, dt, dx, dy)
    dij  = view(u, 2:M-1, 2:M-1)
    di1j = view(u, 1:M-2, 2:M-1)
    dij1 = view(u, 2:M-1, 1:M-2)
    di2j = view(u, 3:M  , 2:M-1)
    dij2 = view(u, 2:M-1, 3:M  )
end
```

The efficiency of the code with stencil depends on how we store the array for u. Since the values of u at a time step n only rely on values from step n=1, Overlapping to all values of the array, Winding numbers in array and slide that window and then apply stencil throw multiply the number by over array with coefficient that come with stencil add them all up and that gives us the $u^2_{i,j}$ value.

2.3 Numerical integration methods

The terms "explicit" and "implicit" are frequently used to describe numerical solution approaches. The computation is said to be explicit when the dependent variables may be directly computed in terms of known quantities. The numerical method is implicit when the dependent variables are described by coupled sets of equations and the solution requires either a matrix or an iterative procedure [16].
The main advantage of implicit solution methods, which are more difficult to implement and demand more computational work in each solution step, is that they

allow for larger time steps. A basic qualitative model will be used to demonstrate how this works.
In brief, the advance of the pressure step in explicit methods must be limited to less than one computational cell per time step. Implicit approaches, on the other hand, use an iterative solution to connect all the cells and allow pressure signals to be conveyed over a grid, so we will use explicit iterative methods to solve our heat equation problem.

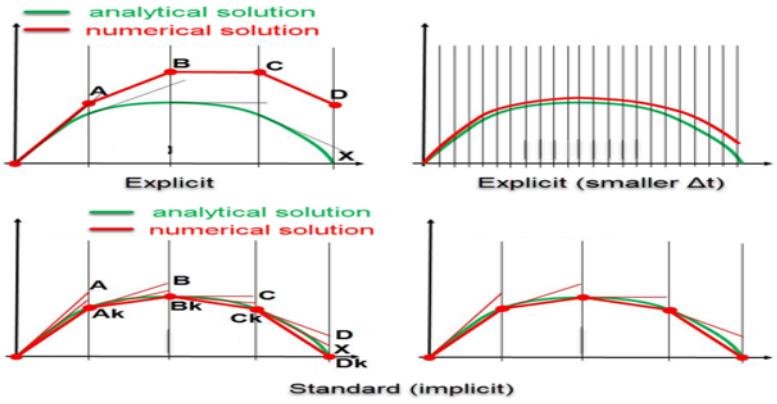


Figure 2.3: Explicit and Implicit graphs [17]

2.3.1 Euler method

Numerical methods can be used to approximate difficult–to–solve differential equations. look at one numerical method called the Euler's Method. Euler's method uses the readily available slope information to start from the point (x_0, y_0) then move from one point to the next (x_1, y_1) along the polygon approximation of the graph of the particular differential equation to ultimately reach the terminal point (x_n, y_n) As illustrated in the diagram 2.4.

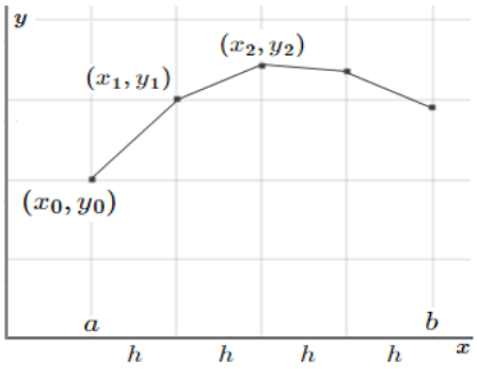


Figure 2.4: Euler method graph [18]

Although interested in determining all of the points along with the differential equa-

tion, it is often the case that the value of y_n at the terminal point is of most interest. More specifically, let $y = f(x)$ be the solution to the differential equation.

$$\frac{dy}{dx} = g(x, y) \quad \text{with} \quad y(x_0) = y_0 \tag{2.35}$$

for $x_0 \leq x \leq x_n$, let $x_i + 1 = x_i + h$, where $h = \dfrac{x_n - x_0}{n}$ and

$$y_{i+1} = y_i + y(x_i, y_i)h, \tag{2.36}$$

or $0 \leq i \leq n - 1$, then

$$f(x_{i+1}) \approx y_{i+1} \tag{2.37}$$

2.3.2 Midpoint method

In numerical analysis, a branch of applied mathematics, the midpoint method is a one-step method for numerically solving the differential equation,

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0. \tag{2.38}$$

The explicit midpoint method is given by the formula

$$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}f(t_n, y_n)\right). \tag{2.39}$$

2.3.3 Runge-Kutta methods

In the forward Euler method, we used the information on the slope or the derivative of y at the given time step to extrapolate the solution to the next time step. Runge-Kutta is a collection of methods. The n-order case requires s free parameters, one for each stage of each implementation.

$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.40}$$

where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}),$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}),$$

$$k_4 = f(t_n + h, y_n + hk_3).$$

k_1 : is the slope at the beginning of the interval, using y (Euler's method).
k_2 : is the slope at the midpoint of the interval, using y and k_1.
k_3: is again the slope at the midpoint, but now using y and k_2.
k_4 : is the slope at the end of the interval, using y and k_3.

Chapter 3

Hardware and Software tools

In this chapter, we focus on the hardware and software tools that we use and the packages that fulfill our needs to overcome the heat equation problem. In section 3.1 is a brief introduction to GPUs, Section 3.2 demonstrates how to use CUDA in C++ and Julia programming language to write a program that solves the two-dimensional heat equation on a GPU core. While at the end of this chapter, sections 3.3 and 3.4 show how to use the DifferentialEquation.jl and DiffEqGPU.jl packages.

3.1 GPU Core

There was no need for a graphics card in the past when computers were only utilized for basic tasks. However, as the popularity of computer gaming grew, there was a huge need for graphics cards. Games today demand a massive quantity of graphics that can't be delivered by integrated graphics alone; of course, the graphics card is the most vital component for any gamer.
A graphics card is not only useful for gaming, but it is also useful for a variety of other tasks. This is especially useful for graphic design editing. These cards, however, are not without flaws. It's vital to weigh the benefits and drawbacks before installing a graphics card. Much like a motherboard contains a central processing unit, a graphics card refers to an add-in board that incorporates the graphics processing chip or unit. This board also includes the raft of components required to both allow the GPU to function and connect to the rest of the system figure 3.1.

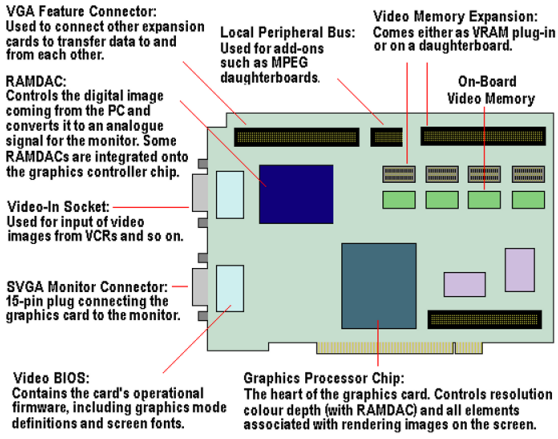


Figure 3.1: Graphic Card Components [19]

Some of the most exciting applications for GPU technology involve AI and machine learning. Because GPUs incorporate an extraordinary amount of computational capability, they can deliver incredible acceleration in workloads that take advantage of the highly parallel nature of GPUs, such as image recognition. Many of today's deep learning technologies rely on GPUs working in conjunction with CPUs.

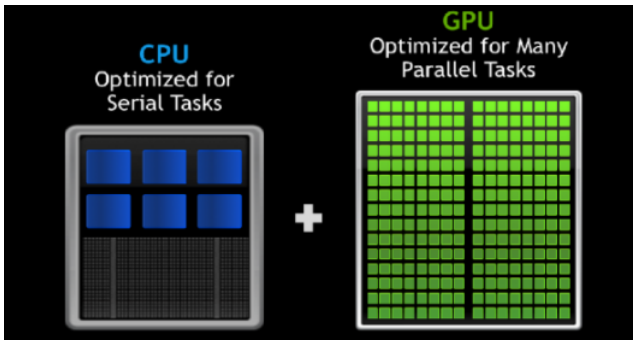


Figure 3.2: CPU vs GPU

In figure 3.2 the CPU on the left has several cores that share the same memory. The GPU on the right has hundreds of cores, but all the cores share one device memory. The main difference between CPU and GPU architecture is that a CPU is designed to handle a wide range of tasks quickly (as measured by CPU clock speed) but is limited in the concurrency of tasks that can be running. Here is a small comparison between the GPU and the CPU In Table 3.1.

CPU	**GPU**
Really fast caches (great for data reuse)	Lots of math units
Fine branching granularity	Fast access to on-board memory
Lots of different processes/threads	Run a program on each fragment/vertex
CPUs are great for task parallelism	GPUs are great for data parallelism
CPU optimised for high performance on sequential codes (caches and branch prediction)	GPU optimised for higher arithmetic intensity for parallel nature (Floating point operations)

Table 3.1: Comparison between CPU and GPU

Graphic processing unit is designed to quickly render high-resolution images and video concurrently. Because GPUs can perform parallel operations on multiple sets of data, they are also commonly used for non-graphical tasks such as machine learning and scientific computation. Designed with thousands of processor cores running simultaneously, GPUs enable massive parallelism where each core is focused on making efficient calculations.
It is a massively parallel architecture. Many problems can be solved quickly and efficiently with GPU computing. A considerable number of arithmetic capabilities are available on GPUs. They boost the pipeline's programmability.

Each GPU currently comes with up to four gigabytes of graphics double data rate (GDDR) DRAM, referred to as global memory in figure 3.3. The GPU's RAM is distinct from the CPU's since they serve as frame buffer memories for rendering graphics.

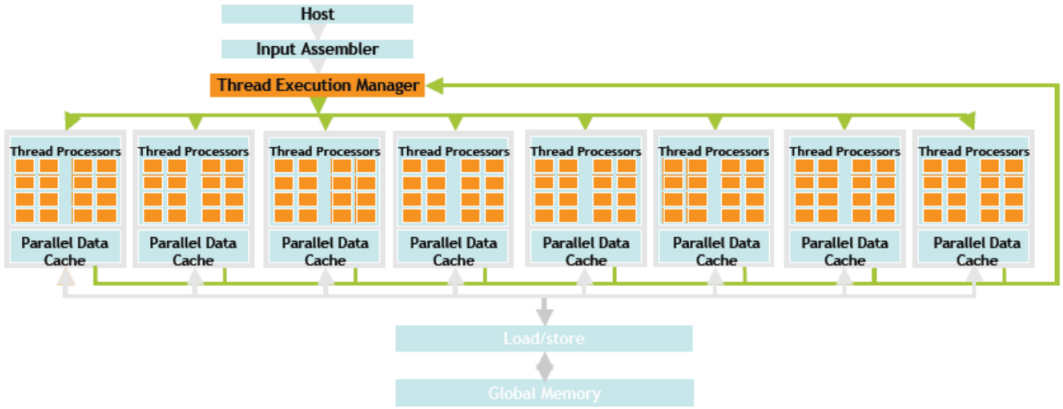


Figure 3.3: GPU Block Diagram [20]

The architecture of a typical CUDA-capable GPU, It can be seen to be an array of streaming processors capable of a high degree of threading.
In figure 3.4 SMs form a building block. However, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation.
Also, Every SM has a number of streaming processors (SPs) that share control logic and instruction cache.

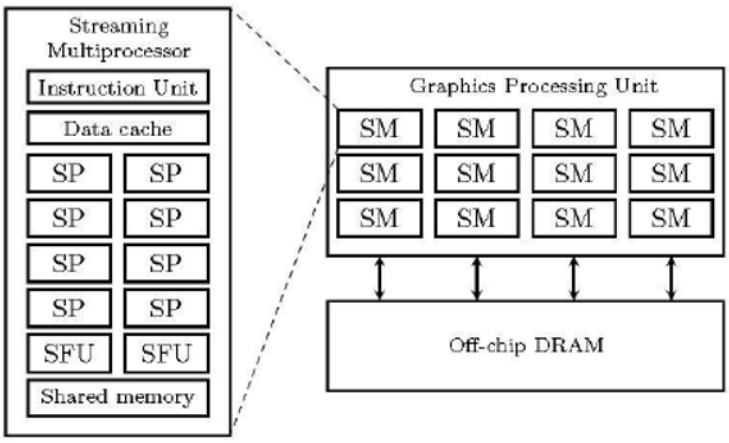


Figure 3.4: schematic of NVIDIA GPU architecture [21]

They carry video pictures and texture information for three-dimensional rendering in graphics applications, but they also serve as very-high-bandwidth, off-chip memory for computing, though with a slightly greater delay than standard system memory. The larger bandwidth compensates for the long delay in massively parallel applications.

3.1.1 NVDIA GTX 1050 Ti

Nvidia's GeForce 10 series of graphics processing units are based on the Pascal microarchitecture, which was first introduced in 2014. This design series superseded the GeForce nine hundred series and was followed by the Turing microarchitecture-based GeForce 16 and GeForce 20 series. In march 2014, the Pascal microarchitecture, named after Blaise Pascal, was revealed as the Maxwell microarchitecture's replacement. The GeForce GTX 1080 and 1070, the series' first graphics cards, were announced on May 6, 2016, and released a few weeks later, on may 27 and June 10, respectively. The architecture uses either TSMC's 16 nm FinFET technology or Samsung's 14 nm FinFET technology. Initially, only TSMC's 16 nm technology was used, but later chips were fabricated using Samsung's newer 14 nm processes (GP107 and GP108).

Figure 3.5: GTX 1050 Ti

The GTX 1050 Ti officially launched on 25^{th} Oct.2016 under the code name GP107-300-A1 and size 132 mm^2. It has a core configuration of 768:48:32 and a core speed of 1290 MHZ for the base core clock and 1392 MHZ for the boost core clock. The bandwidth of the memory is 112 GB/s and the bus width is 128 bits with the GDDR5 type. 1981 (2138) (boost) is a single processing power and 62 (72) (boost) is double processing power. The GTX 1050 has been reduced to 2 GB GDDR5, while the GTX 1050 Ti is even more generously equipped with 4 GB GDDR5 than the 3 GB variant of the GeForce GTX 1060.

Model	**Memory (GB/s)**	**Processing power (boost)**
GT 1030(DDR4)	48	29(35)
GTX 1050 Ti	112	62(67)
GTX 1060	160	120(137)

Table 3.2: Comparison between different NVIDIA models

3.2 CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA's GPU architecture featured in the GPU cards, positioning itself as a new means for general-purpose computing with GPUs. The CUDA architecture is made up of three main pieces that allow the programmer to fully utilize the graphics card's computational capabilities. The CUDA architecture splits the device into grids, blocks and threads in a hierarchical structure as shown in figure 3.6. Since there are a number of threads in one block, a number of blocks in one grid and a number of grids in one GPU, the parallelism that is achieved using such a hierarchical architecture is immense.

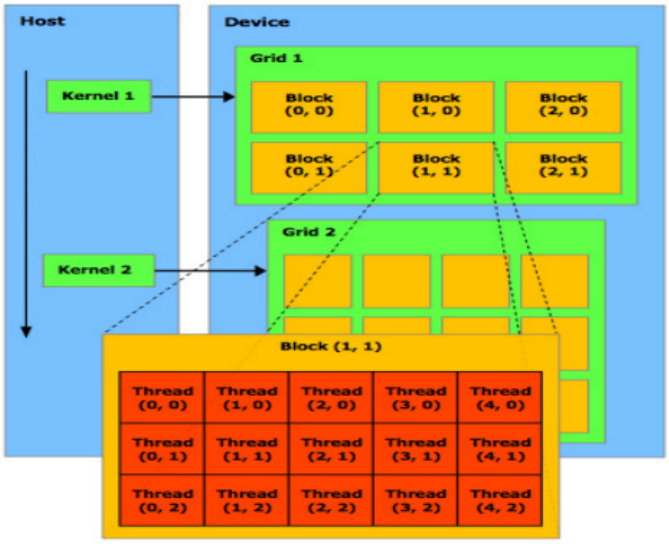


Figure 3.6: Schematization of CUDA architecture [22]

A grid is a collection of threads that all execute the same kernel. There is no synchronization between these threads. Every CUDA call from the CPU is routed through a single grid. Asynchronous action starts a grid on the CPU, although numerous grids can run at the same time. Grids cannot be shared between GPUs in multi-GPU systems since they require many grids for maximum efficiency.
Blocks are used to make grids. Each block is a logical unit with a set of coordinating threads and a certain quantity of shared memory. Blocks are not shared between multiprocessors in the same way that grids are not shared between GPUs. The same program is used by all blocks in a grid. To identify the current block, utilize the built-in variable "blockIdx." Block IDs can be one or two-dimensional (based on grid dimension). A GPU typically contains 65,535 blocks in each dimension.
Threads are used to make blocks. Threads execute on the multiprocessor's individual cores, but unlike grids and blocks, they are not limited to a single core. Each thread, like blocks, has a unique identifier (threadIdx). Thread IDs (depending on block dimension) can be 1D, 2D, or 3D. The thread id is related to the block in which it is located. The amount of register memory available to threads is limited. In most cases, each block can have up to 512 threads. But as of March 2010, with compute capability 2.x and higher, blocks may contain up to 1024 threads.

Figure 3.7 shows how CUDA threads can access data from multiple memory spaces during execution. Each thread has its own private memory. Each thread block has a shared memory that is visible to all block threads and has the same lifetime as the block. The global memory is shared by all threads.

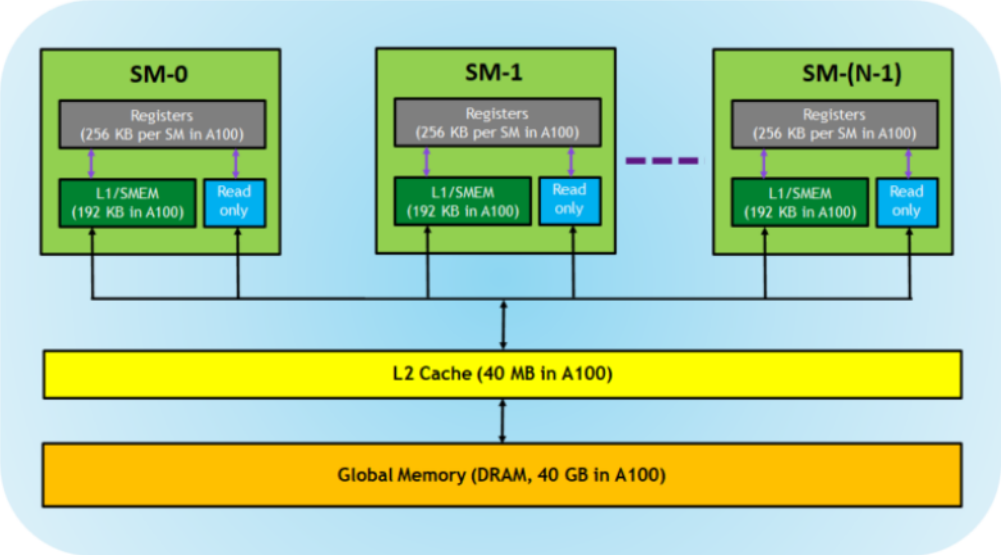


Figure 3.7: CUDA Memory hierarchy [23]

The following memories are exposed by the GPU architecture.
Registers: These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.
Global memory: Is a memory that can both read and write. It's uncached and sluggish, requiring sequential and aligned 16-byte reads and writes (coalesced read-/write).
L1/Shared memory (SMEM): every SM has a fast, on-chip scratchpad memory that can be used as an L1 cache and shared memory. All threads in a CUDA block can share shared memory, and all CUDA blocks running on a given SM can share the physical memory resource provided by the SM.
L2 cache: is shared across all SMs, so every thread in every CUDA block can access this memory. The NVIDIA A100 GPU has increased the L2 cache size to 40 MB, as compared to 6 MB in V100 GPUs.
Read-only memory (ROM): each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.

3.2.1 C++ Programming Model

Figure 3.8 shows the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a co processor to the host running the C++ program. This is the case, for example, when the kernels execute on a GPU and the rest of the C++ program executes on a CPU.
The CUDA programming model also assumes that both the host and the device

maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA run time. This includes device memory allocation and deal location as well as data transfer between host and device memory. To execute any CUDA program, there are three main steps. The first is the host-to-device transfer process, which copies data from host memory to device memory. Then load the GPU program and execute it, caching data on-chip for performance. The last step is the device-to-host transfer process of copying results from device memory to host memory.

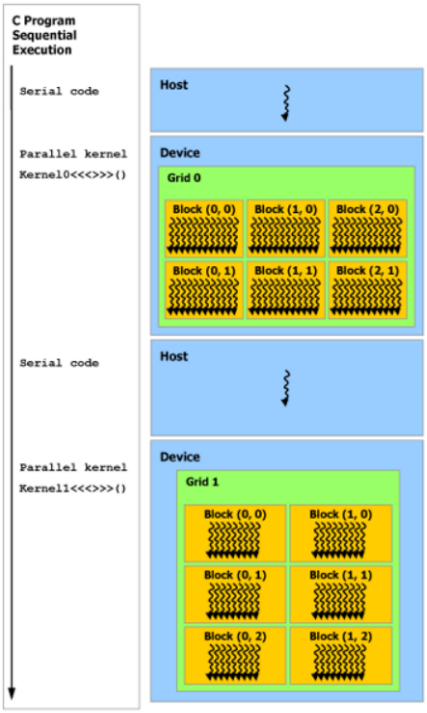


Figure 3.8: Heterogeneous Programming [4]

Every CUDA kernel starts with a global declaration specifier. By using built-in variables, programmers provide a unique global ID to each thread. For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The CUDA programming architecture creates a heterogeneous environment in which the host code executes a C/C++ program on the CPU while the kernel executes on a physically distinct GPU device. The CUDA programming paradigm also assumes that the host and device have their own memory spaces, which are referred to as host memory and device memory, respectively. To make programming easier, CUDA exposes several built-in variables and offers multi-dimensional indexing flexibility. CUDA also manages registers, shared memory, L1 cache, L2 cache, and global memory, among other things. Some of this memory can be used effectively by advanced developers to optimize the CUDA program.

3.2.2 Julia Programming Model

CUDA.jl is a github organization created to unify the many packages for programming graphic processing units in Julia. With its high-level syntax and flexible compiler, Julia is well-positioned to productively program hardware accelerators like GPUs without sacrificing performance.
The Julia CUDA stack requires that users have a working NVIDIA driver and a CUDA toolkit which is needed to develop GPU-accelerated applications. Toolkit includes GPU-accelerated libraries, a compiler, development tools, and the runtime.
To use CUDA.jl, we need a computer with a compatible GPU and to have installed CUDA. we should also install the following packages using Julia's package manager.

```
pkg> add CUDA
pkg> test CUDA
```

The simplest method to make use of the GPU's tremendous parallelism is to define operations as arrays. CUDA.jl has an array type, CuArray, as well as several specialized array operations that run quickly on the GPU hardware. CuArrays are fully functional GPU arrays that can significantly outperform traditional arrays without requiring any code changes. Julia completely implements CuArrays, making the implementation both elegant and generic.

```
A = CuArrayInt(undef,1024)
```

A is a matreix with undefined values with 1024 cells Primarily, it is used to manage GPU memory, and copy data from and back to the CPU.

```
B = zeros(M,M)
B-GPU = CuArray(convert(Array{Float32}, B))
```

The CPU can assign jobs to the GPU and then go do other stuff (such as assigning more jobs to the GPU) while the GPU completes its tasks. Wrapping the execution in a CUDA.@sync block will make the CPU block until the queued GPU tasks are done, similar to how Base.@sync waits for distributed CPU tasks. Without such synchronization, we'd be measuring the time it takes to launch the computation, not the time it takes to perform the computation. But most of the time, we don't need to synchronize explicitly: Many operations, like copying memory from the GPU to the CPU, implicitly synchronize execution [24].

```
function add-broadcast!(y, x)
   CUDA.@sync y .+= x
   return
end
```

To overcome the cost of launching kernels, CUDA makes it possible to build computational graphs, and execute those graphs with less overhead than the underlying operations [25].

```
A = CUDA.zeros(Int, 1)
A .+= 1
graph = capture() do
A .+= 1
end
```

3.3 DifferentialEquation.jl

This is a suite for numerically solving differential equations written in Julia and available for use in Julia, Python, and R. The purpose of this package is to supply efficient Julia implementations of solvers for various differential equations [26].
To install the package, use the following command inside the Julia REPL

```
Pkg.add("DifferentialEquations")
```

and to load the package

```
using DifferentialEquations
```

If we have a model, This model says that the rate of change is proportional to the current value with 0.98 so we can get through it by

```
f(u,p,t) = 0.98u
```

where p is the parameters and t is the time span and we consider time span on this model from t=0.0 to t=1.0.
As we explained at chapter2 we need an Initial condition

```
u0 = 1.0
tspan = (0.0,1.0)
```

If we want to introduce this model, then we define an *ODEProblem* by specifying this function f, this initial condition $u0$, and this time span as

```
prob = ODEProblem(f,u0,tspan)
```

To solve our ODEProblem we use the command *solve*.

```
sol = solve(prob)
```

Controlling the Solver

DifferentialEquations.jl has a common set of solver controls among its algorithms, which can be found at reference number [27]. We can also turn off all intermediate savings. by using *save_everystep=false*.
To control the tolerance *abstol=1e-xx* and *reltol=1e-xx* are used.
There is a trade-off between accuracy and speed, so determining what the right balance depends on the problem.
when *saveat* is used, the continuous output variables are no longer saved, and thus *sol(t)*, the interpolation, is only first order. As an example *saveat=[0.2,0.7,0.9]* so solver will go only through these values.
If we need to reduce the amount of savings, we can also turn off the continuous output directly via *dense=false*.
There is no best algorithm for numerically solving a differential equation. When we call to solve the problem, DifferentialEquations.jl makes a guess at a good algorithm for our problem, given the properties that we ask for (the tolerances, the saving information, etc.). However, in many cases, we may want more direct control. As an example, we want to use the Euler method to solve one-dimensional heat conduction

```
sol = solve(prob, Euler(), dt=t, save_everystep=false).
```

And for two dimensional heat case

```
sol   = solve(prob, Tsit5(), dt=t, progress=true, save_start=true).
```

3.4 DiffEqGPU.jl

This library is a component package of the DifferentialEquations.jl ecosystem. It includes functionality for making use of GPUs in the differential equation solvers [28].
DiffEqGPU.jl is compatible with all array operations take place on the GPU, including any implicit solves. The native Julia libraries, including (but not limited to) OrdinaryDiffEq, StochasticDiffEq, and DelayDiffEq, are compatible with u0 being a CuArray. When this occurs, all array operations take place on the GPU, including any implicit solves. This is independent of the DiffEqGPU library. These speedup the solution of a differential equation which is sufficiently large or expensive. This does not require DiffEqGPU.jl. To insert the DiffEqGPU package on Julia REPL

```
julia> ]
pkg> add DiffEqGPU
```

To load it

```
using DiffEqGPU
```

Here is other model which defined the problem by using *CuArray*

```
using OrdinaryDiffEq, CUDA, LinearAlgebra
u0 = cu(rand(1000))
A  = cu(randn(1000,1000))
f(du,u,p,t)  = mul!(du,A,u)
```

then define our problem as function

```
f(du,u,p,t)  = mul!(du,A,u)
```

the last step use the *ODEProblem* and *solve* commands to get the results.

```
prob = ODEProblem(f,u0,(0.0f0,1.0f0))    # Float32 is better on GPUs!
sol = solve(prob)
```

Parameter-parallel GPU methods are provided for the case where a single solve is too cheap to benefit from within-method parallelism, but the solution of the same structure is required for very many different choices of u0 or p.
For these cases, DiffEqGPU exports the following ensemble algorithms EnsembleGPUArray which it utilizes the CuArray setup to parallelize ODE solves across the GPU and EnsembleCPUArray is a test version for analyzing the overhead of the array-based parallelism setup.

Chapter 4

Implementation

In the current chapter, Implementation of heat conduction computations at both cores are described. In section 4.1, simulation of the semi-discretized one-dimensional heat equation with Dirichlet and Neumann boundary conditions is discussed. In section 4.2, simulation of the full and semi-discretized two-dimensional heat equations is also performed, with both boundary conditions.

4.1 One-dimensional heat equation

By applying a heat cell with a temperature of 500 K somewhere on a homogeneous rod with length L, we need to discretize the rod into a grid as we explained in chapter 2. The characteristics of the rod are shown in table 4.1.

Parameters	**Symbol**	**Value**
Diffusivity	α	0.0001 m^2/s
Length	L	0.5 m
Number of grid points	M	66 points
x-grid spacing	Δx	7.69 mm
Largest stable time step	Δt	2.95 s
Heat cell	u[28:38]	500 k

Table 4.1: Simulation Parameters

4.1.1 CPU computation by semi-discretization

We define M-matrix with a zero value on the CPU core. Then we apply the heat source someplace on the heat matrix.
By creating a function that has the role of applying a stencil computation, which windows all nodes, slides all of them, and applies a pattern (the diffusion of the heat cell) and set the boundary condition. We know that the one-dimensional heat equation is not a massive equation, that is the reason which makes us leave the DifferentialEquations package choose the algorithmic method to solve our problem. Because we know that the one-dimensional heat equaftion is not a massive equation,

we let the DifferentialEquations package choose the best solver for us.

```
julia> sol.alg
CompositeAlgorithm{Tuple{Tsit5{typeof(OrdinaryDiffEq.trivial_limiter!), typeof(OrdinaryDiffEq.trivial_limiter!), Static.False}, Rosenbrock23
{11, false, GenericFactorization{LinearSolve.RFWrapper{true, true}}, typeof(OrdinaryDiffEq.DEFAULT_PRECS), Val{:forward}, true, nothing}}, O
rdinaryDiffEq.AutoSwitchCache{Tsit5{typeof(OrdinaryDiffEq.trivial_limiter!), typeof(OrdinaryDiffEq.trivial_limiter!), Static.False}, Rosenbr
ock23{0, false, Nothing, typeof(OrdinaryDiffEq.DEFAULT_PRECS), Val{:forward}, true, nothing}, Rational{Int64}, Int64}}((Tsit5(stage_limiter!
 = trivial_limiter!, step_limiter! = trivial_limiter!, thread = static(false)), Rosenbrock23{11, false, GenericFactorization{LinearSolve.RFW
rapper{true, true}}, typeof(OrdinaryDiffEq.DEFAULT_PRECS), Val{:forward}, true, nothing}(GenericFactorization{LinearSolve.RFWrapper{true, tr
ue}}(LinearSolve.RFWrapper{true, true}()), OrdinaryDiffEq.DEFAULT_PRECS)), OrdinaryDiffEq.AutoSwitchCache{Tsit5{typeof(OrdinaryDiffEq.trivia
l_limiter!), typeof(OrdinaryDiffEq.trivial_limiter!), Static.False}, Rosenbrock23{0, false, Nothing, typeof(OrdinaryDiffEq.DEFAULT_PRECS), V
al{:forward}, true, nothing}, Rational{Int64}, Int64}(28, 0, Tsit5(stage_limiter! = trivial_limiter!, step_limiter! = trivial_limiter!, thre
ad = static(false)), Rosenbrock23{0, false, Nothing, typeof(OrdinaryDiffEq.DEFAULT_PRECS), Val{:forward}, true, nothing}(nothing, OrdinaryDi
ffEq.DEFAULT_PRECS), true, 10, 3, 9//10, 9//10, 2, false, 5))
```

Figure 4.1: The solution algorithm

We can see from a figure 4.1, that the solver used two methods, the Tsit5 and Rosenbrock23, as the best choices to solve our problem. It is a good choice for very large systems (greater than 1000 ODEs).
The time span is limited from zero to one hundred mile seconds.

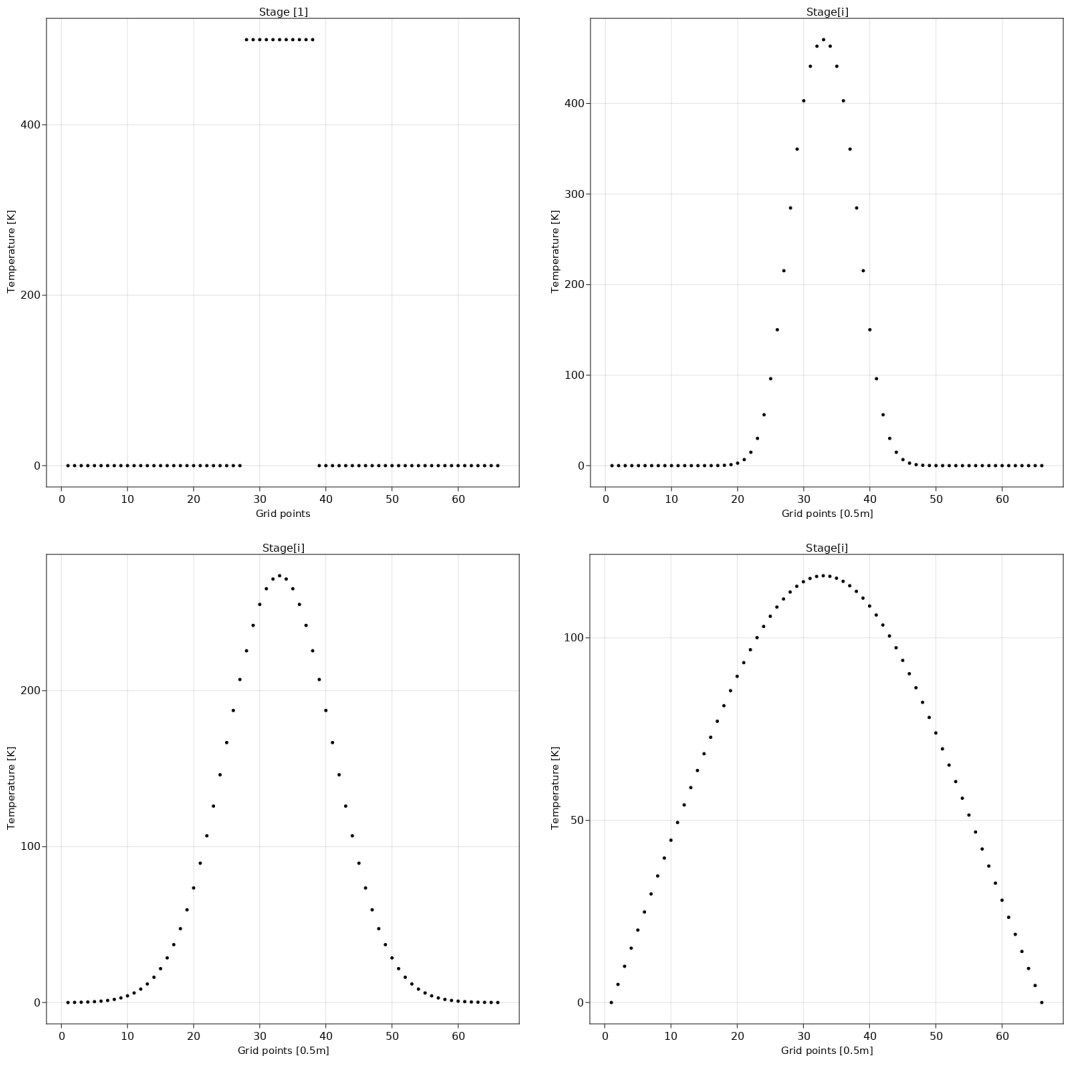


Figure 4.2: One-Dimensional Semi-Discretization on CPU with DBC

Each graph describes a stage in the heat diffusion system. Starting from stage one, which shows applying the heat cell suddenly, and by going throw the stages one by one we can see how the heat spread along the rod.
We notice how the boundary of rods stays constant according to the Dirichlet boundary condition, where we fixed the temperature to be zero kelvin at the beginning and end of the rod. The results are visualized by the Makie.jl package. Makie.jl is a data visualization ecosystem for the Julia programming language, with high performance and extensibility [29].

By Changing the boundary condition from a fixed boundary condition to a Neumann boundary condition

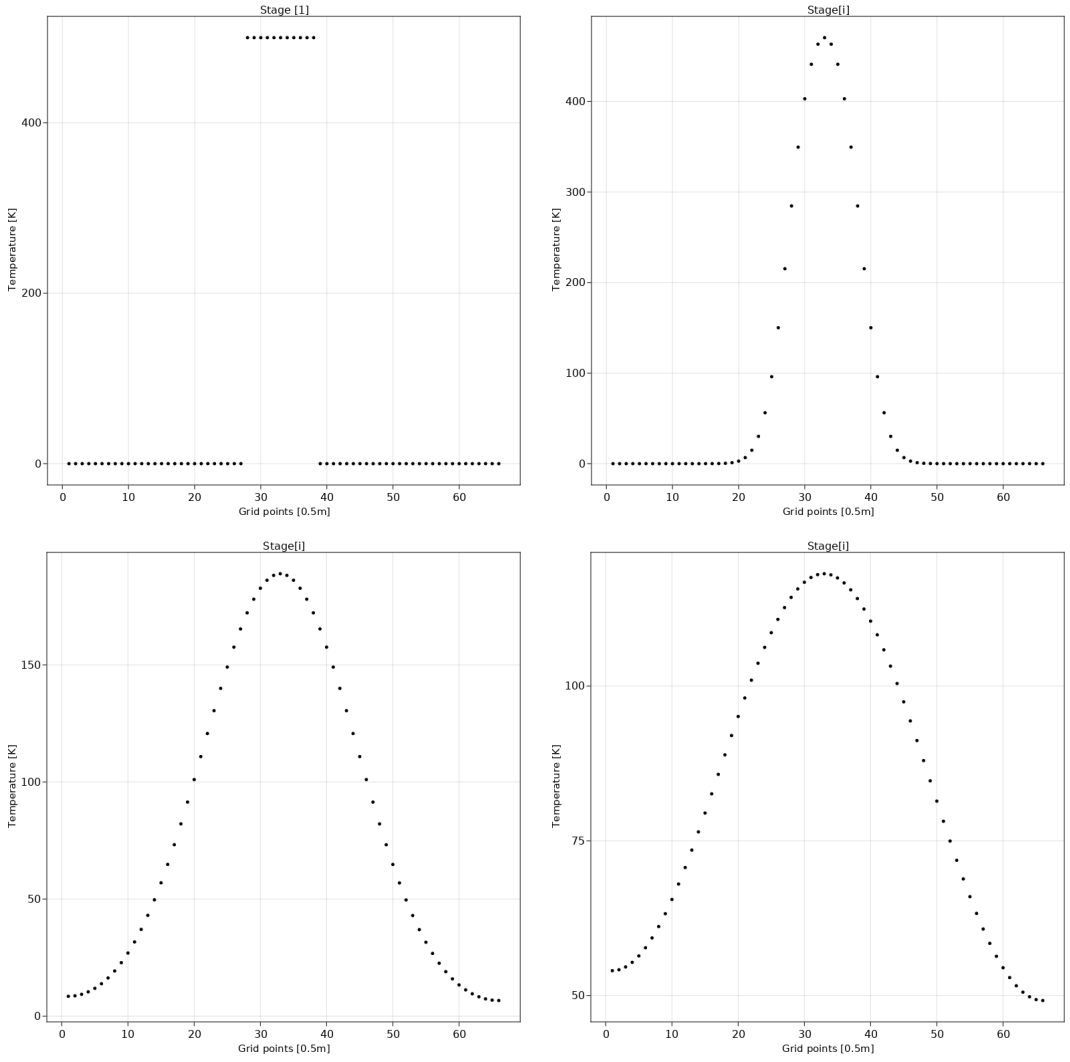


Figure 4.3: One-Dimensional Semi-Discretization on CPU with NBC

As we can observe the change on the boundaries of the rod, it started from a temperature of zero kelvin until it reached a temperature of fifty-nine kelvin on the left side of the rod and forty-eight kelvin on the right side. Graph is increased from the left side than the right side as the heat cell is given at the end of the rod not at the center. As from sixty sixty steps, the heat was applied from twenty-eight to thirty-eight and the reason is that we wanted to see the difference clearly.

4.1.2 GPU computation by semi-discretization

Computations of the system are switched from the central processing unit to the graphic processing unit. To access the GPU, we defied our array as a float 32 cu-array. It is recommended to use Float32 consistently for the dependent variable, parameters, and time on the GPU, especially on consumer-grade hardware.

```
u=CUDA.zeros(M)
u_GPU[28:38] .= 500f0
```

Creating the system using *CUDA.zeros* and defining our heat cell as a temperature of 500 degrees kelvin somewhere on the rod By Selecting the the time span to be $tspan = (0f0, 100f - 2)$ and using both package DifferentialEquations and DiffEqGPU. Our problem defined as.

```
using DifferentialEquations, DiffEqGPU
prob = ODEProblem(diffuse!, u_GPU, tspan)
sol =solve(prob)
```

Here are the outcomes.

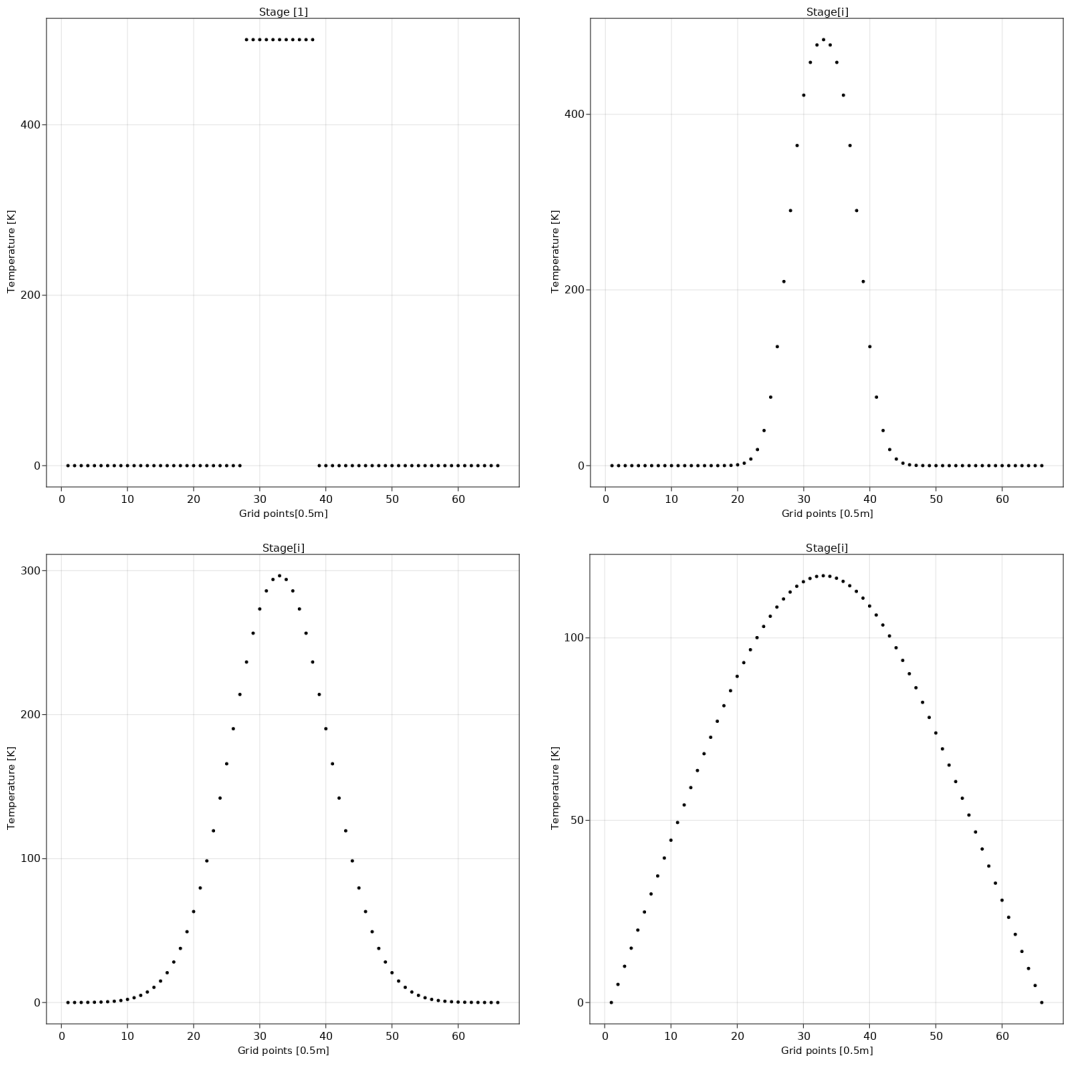


Figure 4.4: One-Dimensional Semi-Discretization on GPU with DBC

After changing the boundary condition from a fixed boundary condition to a Neumann boundary condition, We received the data as follows:

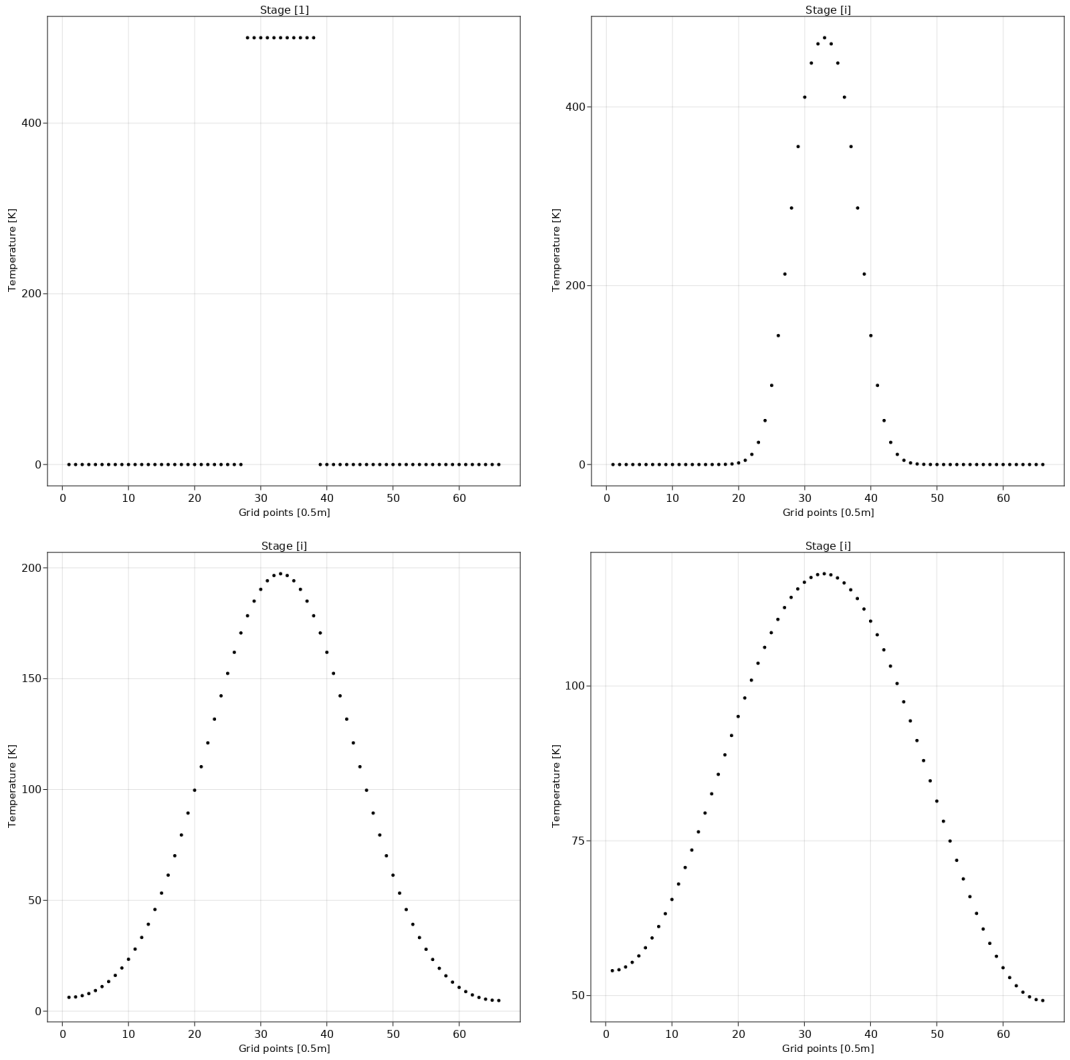


Figure 4.5: One-Dimensional Semi-Discretization on GPU with NBC

As expected, we received the same data from the GPU with the same kind of explicit algorithms. But the variation is in the time that we received the data on it. In this case, the CPU was faster than the GPU. The problem is too small for GPU.

4.2 Two-dimensional heat equation

With the same technique, we used on a one-dimensional case, Applying a heat source with temperature 500 K somewhere on a homogenise surface area with length L and width W. The characteristics of the samples depend on the purpose of the process. We use our samples to do full and semi discretization with Dirichlet and Neumann boundary condition by both cores.

Parameters	**Symbol**	**Sample-one**	**Sample-two**
Diffusivity	α	0.0001 m^2/s	0.0001 m^2/s
Width	W	0.4 m	0.6 m
Length	L	0.5 m	0.8 m
Numbers of grids x-axis	N_x	60 points	60 points
Numbers of grids y-axis	N_y	66 points	80 points
x-grid spacing	Δx	8.47 mm	13.5 mm
y-grid spacing	Δy	6.15 mm	7.59 mm
Largest stable time step	Δt	0.123 s	0.219 s
Heat cell	u[23:35, 28:38]	500 K	500 k

Table 4.2: Simulation Parameters

4.2.1 CPU computation by full-discretization

The sample-one which has a surface area of 0.5 m × 0.4 m. The disassembly is 60 steps in the x-direction and 66 steps in the y-direction. We define a Nx×Ny matrix with a zeros values on CPU core, Applying a the heat source somewhere on the plate as

```
u= zeros(Nx,Ny)
u[23:35, 28:38] .= 500.
```

Run the diffuse function 1000 times, which has the role of applying a stencil computation on the x-axis and y-axis, The for-loop lets the heat spread all over the plate.

```
for i in 1:1000 ; diffuse!(u, a, dt, dx, dy)   end.
```

Set the boundaries of the plate at temperature equal to zero kelvin (Dirichlet boundary condition).

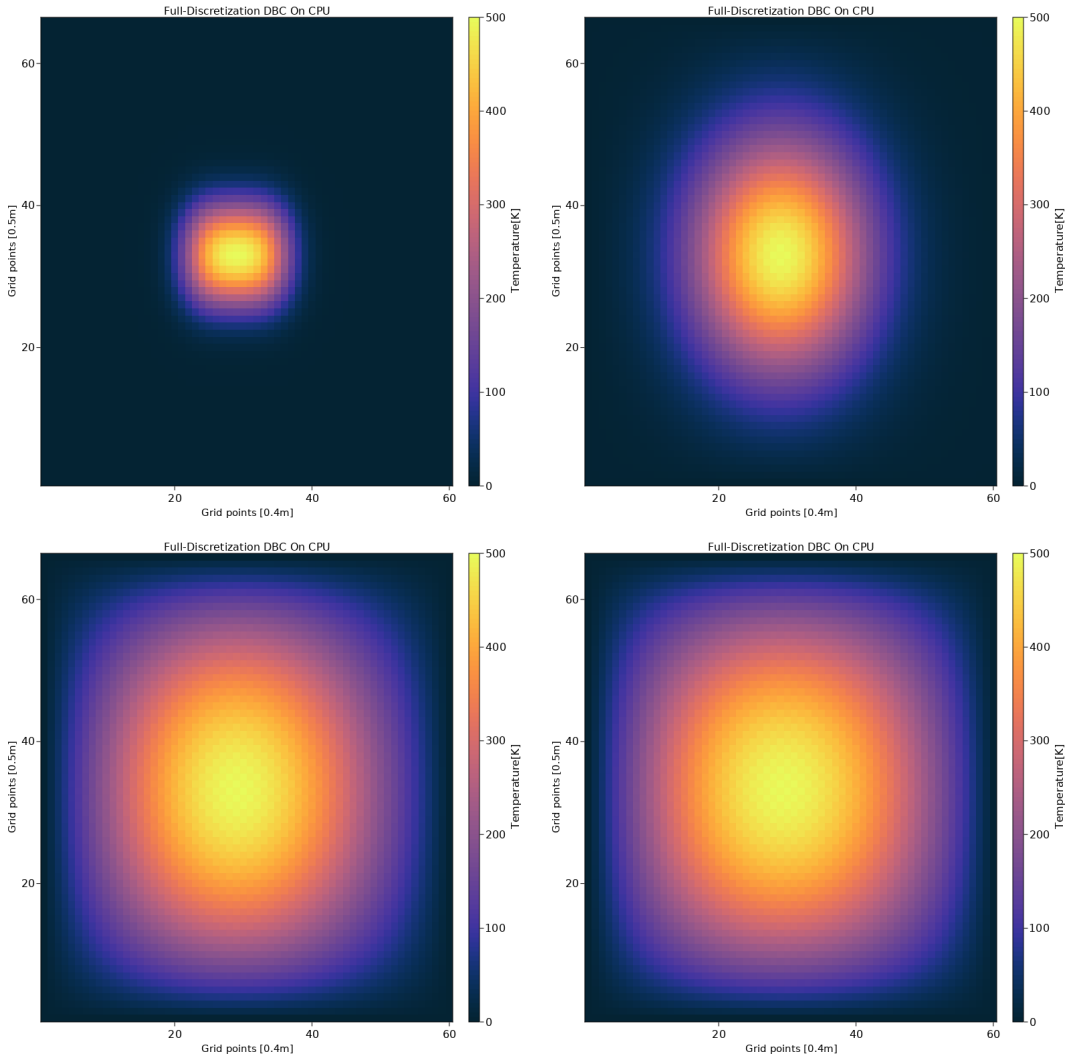


Figure 4.6: Two-Dimensional Full-Discretization on CPU with DBC

We can notice that the heat does not go out of the system. The system is completely isolated from the surrounding atmosphere.
After changing the boundary condition from Dirichlet boundary condition to Neumann Condition and makes our system bigger by increasing the length of the plate. The sample is switched now too second sample.

```
@. u[1, :]  += a * dt * (2*u[2, :] - 2*u[1, :])/dx^2
@. u[Nx, :] += a * dt * (2*u[Nx-1, :] - 2*u[Nx, :])/dx^2
@. u[:, 1]  += a * dt * (2*u[:, 2]-2*u[:, 1])/dy^2
@. u[:, Ny] += a * dt * (2*u[:, Ny-1]-2*u[:, Ny])/dy^2
```

This is part of the diffuse function, The role of it to update boundary condition to Neumann bc after apply diffusion of the heat cells.

The results of full discretization are as follows.

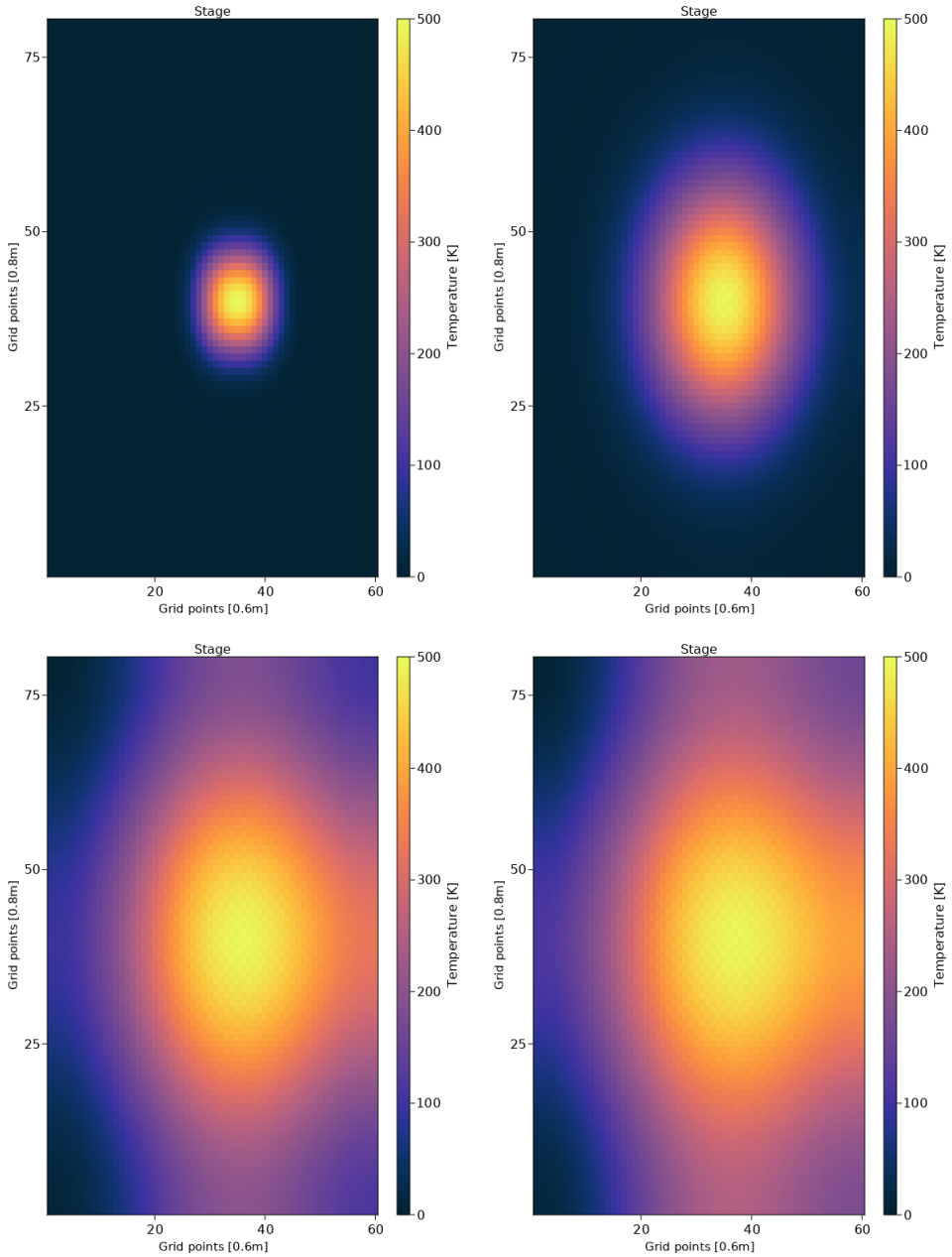


Figure 4.7: Two-Dimensional Full-Discretization on CPU with NBC

It is noticed as the heat goes out of the system. The system is not isolated anymore. Each stage shows how the heat spreads all over the plate. The last stage shows how the heat goes out of the system. It is sifted to the right because the heat cell is applied at range 30:40 in x-direction and from 35 to 45 at y-direction, which is oblique to the right on the x-axis and is at the center of the y-axis.

4.2.2 GPU computation by full-discretization

Here we wanted to transfer our system from the CPU core to the GPU and see if we would receive the same data or not.
We define a zero NX × Ny-matrix on GPU core, applying the heat source somewhere on the with surface area 0.4m × 0.5m as we did on the case of Dirichlet boundary condition on CPU(Sample-one).

```
u_GPU = CUDA.zeros(Nx,Ny)
u_GPU[25:35, 28:38] .= 500
```

Here is the way to explain how we adjust the temperature of outlines to zero kelvin.

```
temp_left   = 0
temp_right  = 0
temp_bottom = 0
temp_top    = 0

@. u[1, :]    = temp_left
@. u[Nx, :]   = temp_right
@. u[:, 1]    = temp_bottom
@. u[:, Ny]   = temp_top
```

The result are shown as follows.

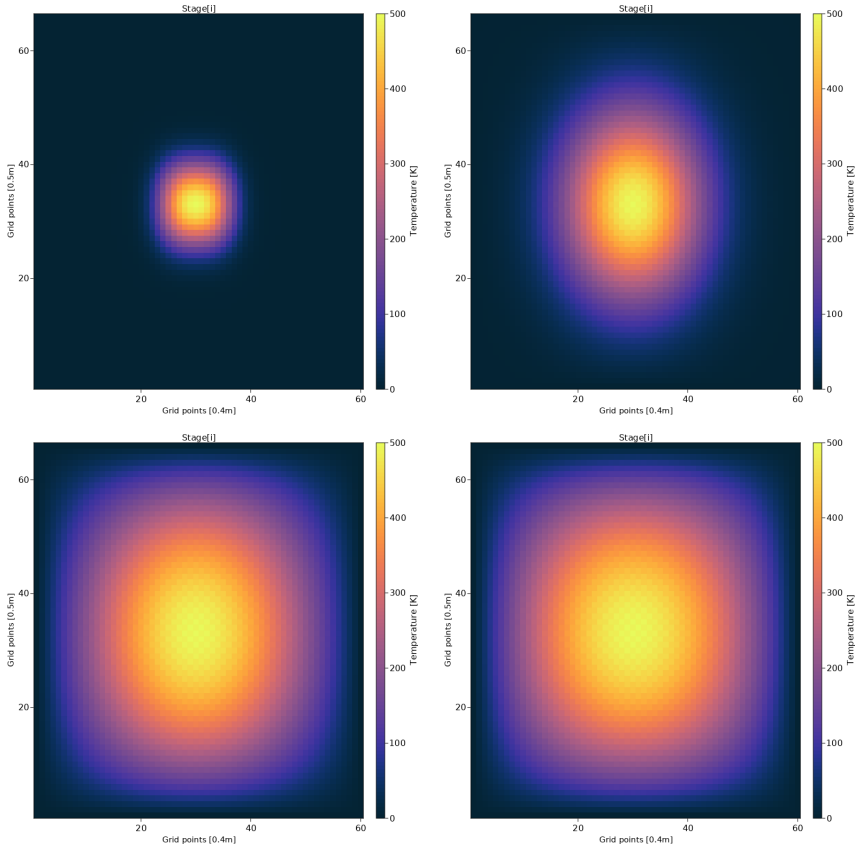


Figure 4.8: Two-Dimensional Full-Discretization on GPU with DBC

We obtain the same implementation as we received it from the center processing

unit by adapting the factors and samples to be identical on both cores.
For Neumann boundary condition. Our model (0.6m × 0.8m) would be set as the CPU case, and the reason that we want to make the compere the results between the data which we received from CPU and and the data which we received it from GPU.

```
W  = 6f-1                          # Width
L  = 8f-1                          # Length
Nx = 60                            # No.of grids in x-axis
Ny = 80                            # No.of grids in y-axis

u_GPU= CUDA.zeros(Nx,Ny)           # Heat matrix
@. u_GPU[30:40, 35:45] = 500       # Heat cell
```

The result are shown as follows.

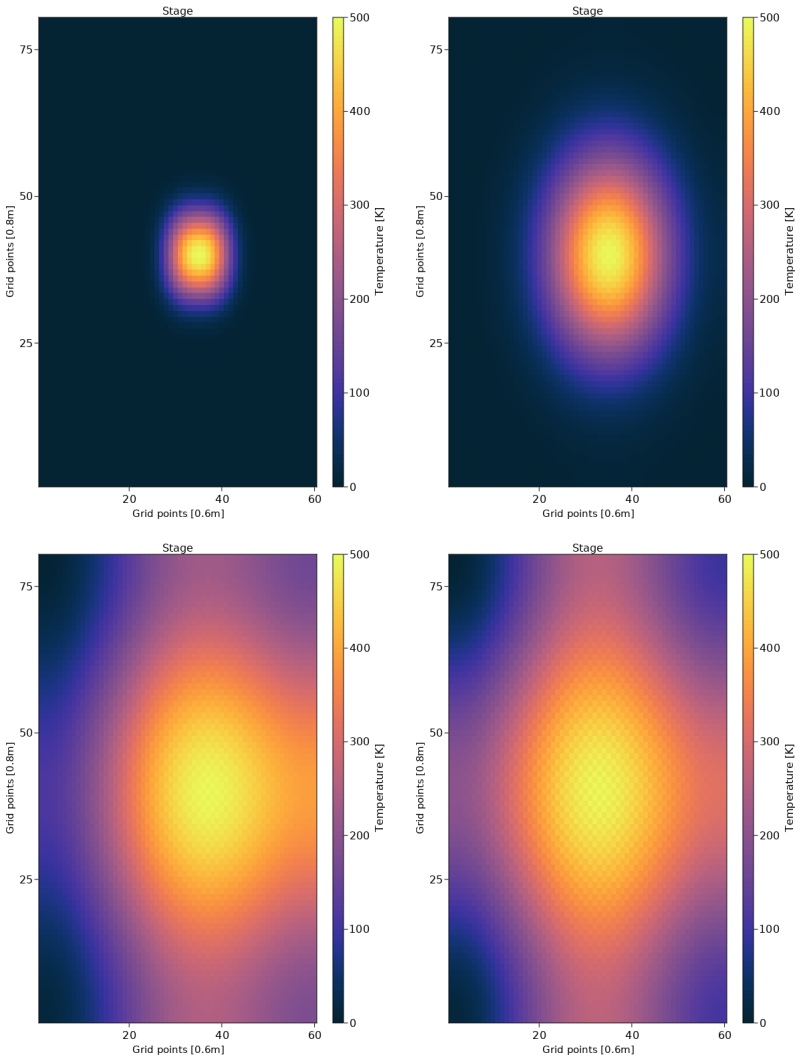


Figure 4.9: Two-Dimensional Full-Discretization on GPU with NBC.

As expected, the results on both cores are similar to the full-discretization without using any packages. The difference is at the boundary of Dirichlet, where the boundary is fixed at zero Kelvin. However, the boundary changes in Neumann boundary condition.

4.2.3 CPU computation by semi-discretization

Semi-discretization proofs it is efficiency than the full-discretization as on CPU we had used many algorithmic method to get our implementation.
Here to code our problem we needed to add other variable to store the process of difusion on it dijij = view(du, 2:Nx-1, 2:Ny-1).
After fixed the boundary condition to Dirichlet boundary condition. we define the problem as prob = ODEProblem(diffuse!, u, tspan) with time span tspan = (0.0, 100.0).
We didn't do our computation manually as in full-Discretization. we used the Euler method which is inserted already at DifferentialEquations package.

```
using DifferentialEquations, DiffEqGPU
tspan = (0.0, 100.0)
prob = ODEProblem(diffuse!, u, tspan)
sol = solve(prob, Euler(), dt=t, save_everystep=false)
```

Here is our result

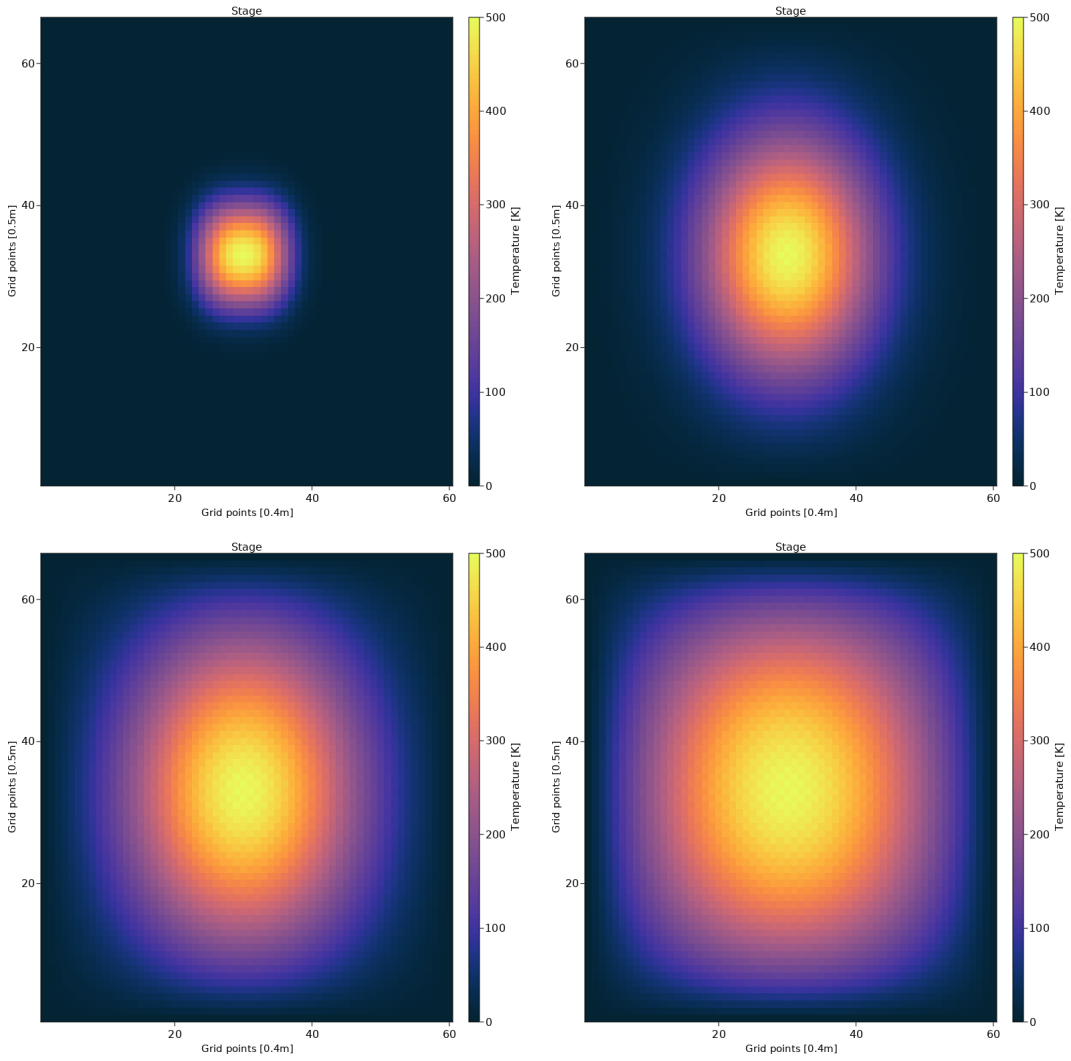


Figure 4.10: Two-Dimensional Semi-Discretization on CPU with DBC

For Neumann boundary condition we used Tsit5 to get the results.

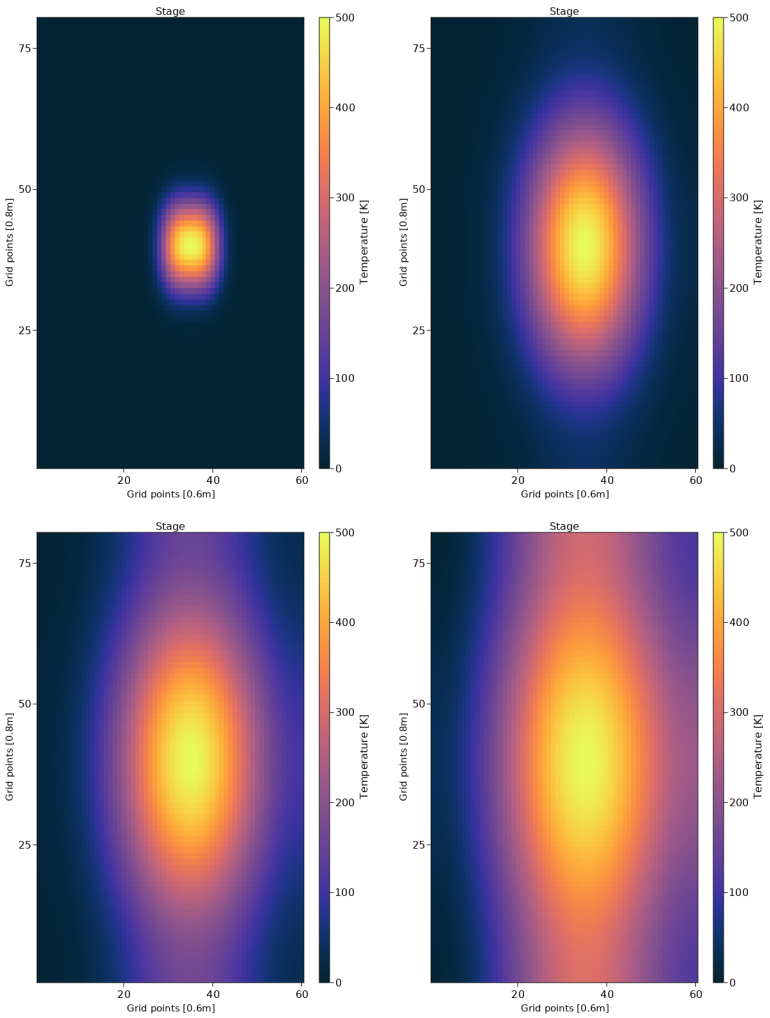


Figure 4.11: Two-Dimensional Semi-Discretization on CPU with NBC

Here are some of our algorithms that run at high efficiency on the CPU.

```
@time sol = solve(prob, Euler(), dt=t)      # 2.65 seconds
@time sol   = solve(prob, Tsit5(), dt=t)    # 1.25 seconds
@time sol = solve(prob,BS3(), dt=t)         # 1.86 seconds
@time sol = solve(prob, Midpoint(), dt=t)   # 1.21 seconds
@time sol = solve(prob,Ralston(), dt=t)     # 1.064 seconds
```

The native OrdinaryDiffEq.jl algorithms are vastly more efficient than the other choices. **Euler** is a first-order explicit Runge-Kutta method solver, A-B-L-stable and explicit FSAL Runge-Kutta method [30]. **Midpoint** is a parallelized explicit extrapolation method. Midpoint extrapolation using Barycentric coordinates [31]. **Ralston** is explicit Runge-Kutta Method. It is optimized second order midpoint method. Uses embedded Euler method for adaptive. **BS3** is a third-order, four-stage explicit FSAL Runge-Kutta method with embedded error estimator of Bogacki and Shampine [32] [33]. **Tsit5** has a high efficiency, when more robust error control is required. It is a fourth-order, five-stage explicit Runge-Kutta method with embedded error estimator of Tsitouras. Free 4th order interpolant [34].

4.2.4 GPU computation by semi-discretization

We transmit our system from CPU core to the GPU and analysis the properties of the system.
After introducing the system to the graphics card, especially the graphic processing unit. We receive this result under the Dirichlet boundary condition.

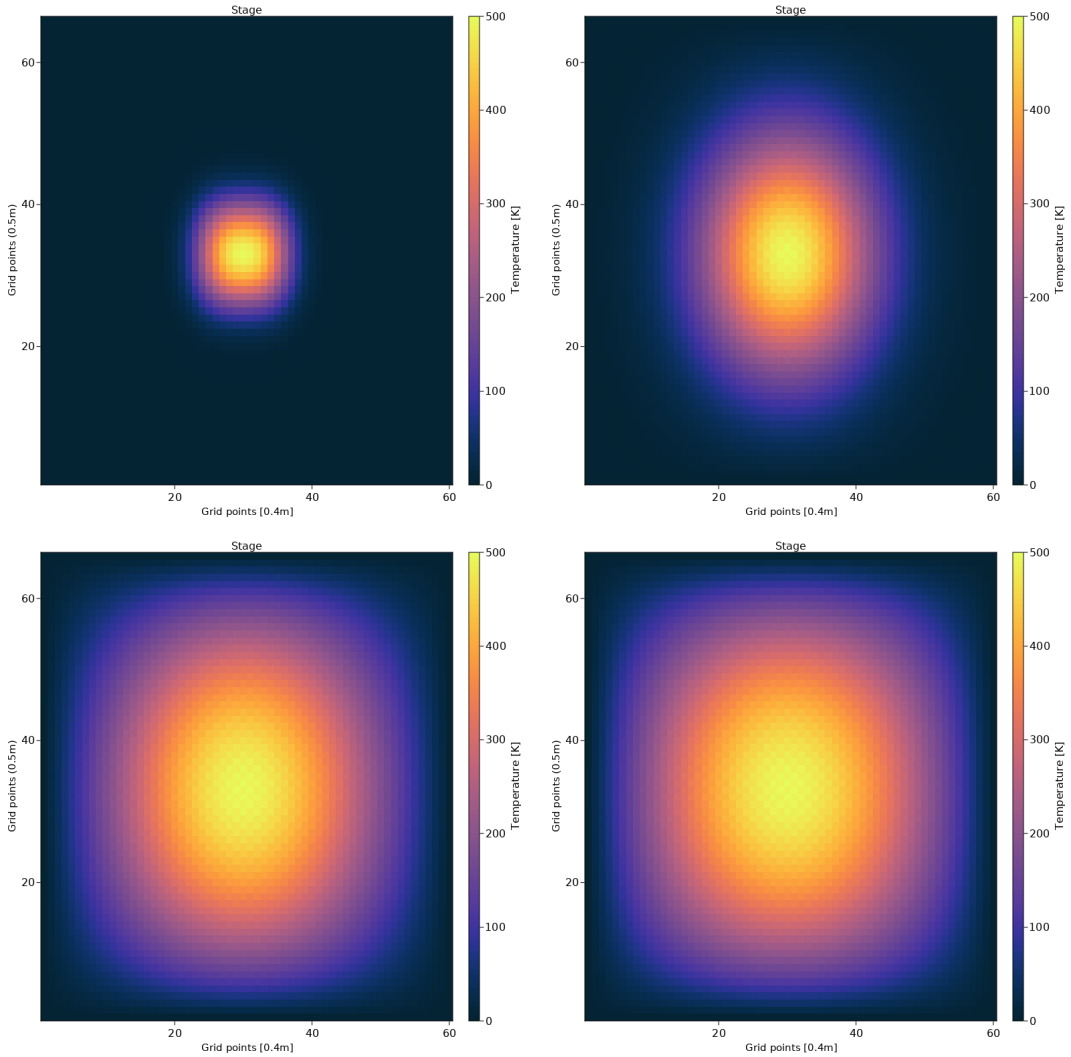


Figure 4.12: Two-Dimensional Semi-Discretization on GPU with DBC

The first stage shows the heat cell without any diffusion yet. The black spots start to change in the second stage from black to dark blue and from dark blue to purple in all directions (360˚ on the surface) till they reach their highest degree, which we refer to as the yellow color. On the third stage, the heat spot will increase without getting out and with time passing till we reach stage number four, where the yellow spot will keep spreading till it covers all the plates without losses on the system. The system is well isolated. We consider this an ideal case that will never happen in real life.

The results are shown as follows for the Neumann boundary condition.

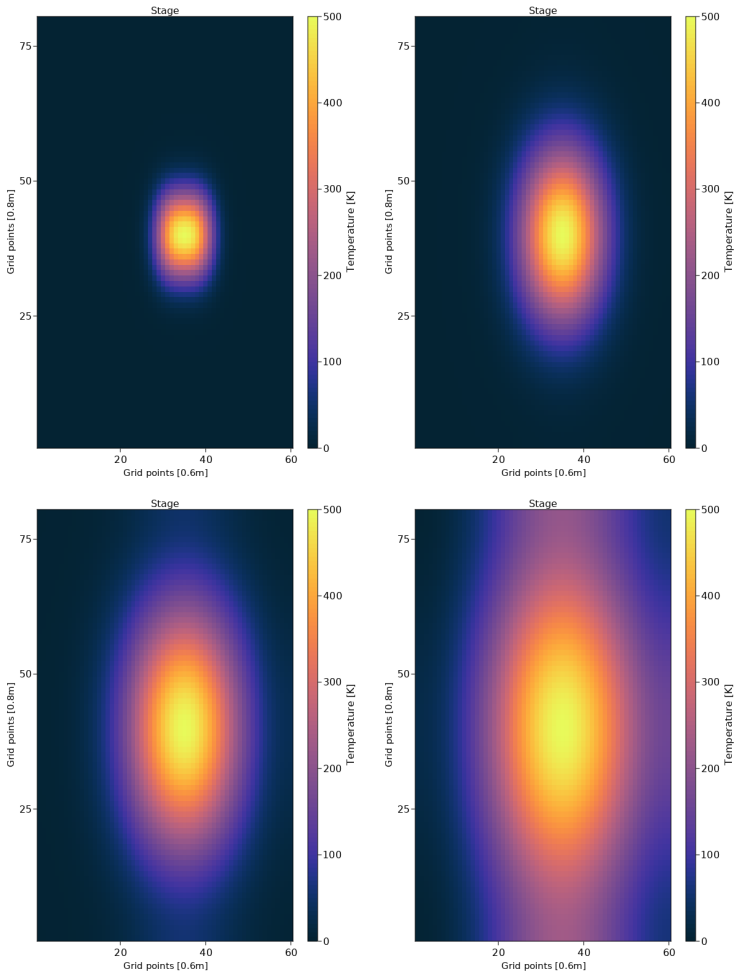


Figure 4.13: Two-Dimensional Semi-Discretization on GPU with NBC

Here is some of our algorithmic methods which is working at a high efficiency on the GPU.

```
@time sol = solve(prob, Euler(), dt=t)           # 1.2 seconds
@time sol = solve(prob, Tsit5(), dt=t)           # 0.94 seconds
@time sol = solve(prob,RK4(), dt=t)              # 1.23 seconds
@time sol = solve(prob, SSPRK22(), dt=t)         # 1.15 seconds
@time sol = solve(prob,AitkenNeville(), dt=t)    # 2.5 seconds
```

For medium accuracy calculations, RK4 is a good choice. **RK4** The canonical Runge-Kutta Order 4 method. Uses a defect control for adaptive stepping and maximum error over the whole interval [30].
SSPRK22 is the two-stage, second order strong stability preserving (SSP) method of Shu and Osher (SSP coefficient 1, free 2nd order SSP interpolant). Fixed timestep only [35]. **AitkenNeville** parallelized explicit extrapolation method Euler, extrapolation using Aitken-Neville with the Romberg Sequence [36].

Chapter 5

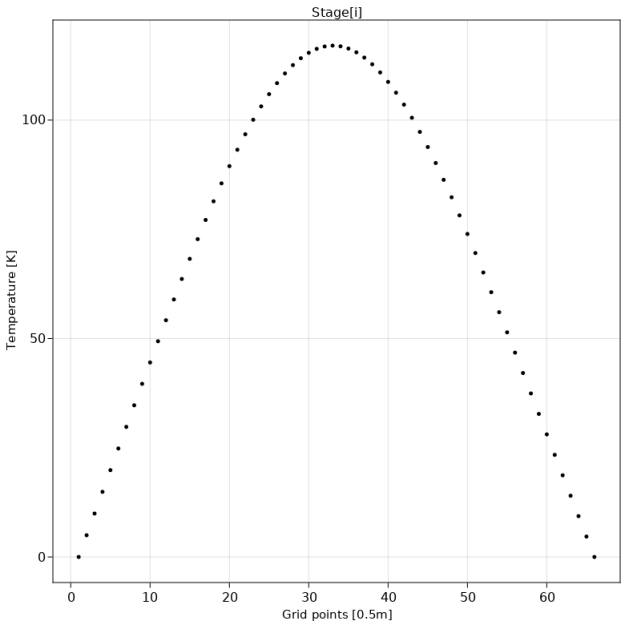
Conclusion

In this research, we were interested in the data processing of heat equations by using a high-performance programming language. We studied heat analysis theory, the finite difference method, and discretization analysis for partial differential equations. Within the scope of the presented work, we tried to work on three axes, First axis to compere between the quality of the result by semi discretization and in full discretization methods. From the result the semi discretization is more efficient numerical method that provides a finite-dimensional matrix approximation of the infinite-dimensional monodromy matrix.
The second axis was the boundaries. Once we specified the boundaries at zero Kelvin, this method is called the Dirichlet boundary condition, which makes the system completely isolated from the surrounded area. In the second case, we used the Neumann boundary condition and sighted how the heat flows out of the system to the temperature median.
The third axis was the computational costs which took place on the mother board, especially at the center processing unit, for the time of computation that happened at the graphic card, especially the graphic processing unit. We reach the conclusion that GPUs have thousands of cores, proving their capability in massive systems that need to do thousands of "stuff" in parallel. So in the one-dimensional case, the actual kernels that are called are slower on the GPU than on the CPU. But when we switched to the two-dimensional case, the cores of the GPU were faster than the CPU's cores.
We used two methods to access and go through each cell in the heat matrix stencil computation and for-for-loop. We get in touch with the Euler and Rung-Kutta methods, which are inserted in the DifferentialEquation.jl package.
The Makie.jl library is used to visualize the results and it is GPU stable.
GPUs have hundreds or thousands of tiny cores dedicated to a single task, such as image processing and data analysis. Face recognition, video surveillance, autonomous driving, automated visual inspection, machine learning, and other edge AI computing applications will all benefit from the GPU computer.
Computing with GPU is a vast subject that requires a plethora of words to express how beneficial it is to our industrial revolution.
The three-dimensional systems are not compatible with the GPU core and with the last version of the code we have. The functionality of the GPU can be tested more by a three-dimensional heat equation as GPUs have an advantage in solving the heat equation.

Appendix A

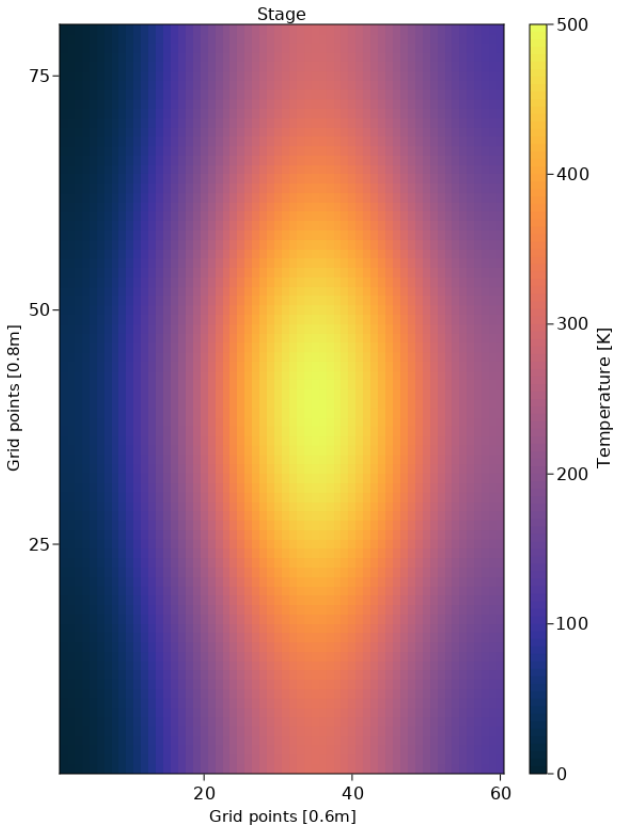
Data Samples

A.1 CPU Analysis




```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.200 ns … 29.600 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.300 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.377 ns ±  0.561 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

Figure A.1: One-dimensional computational costs on CPU



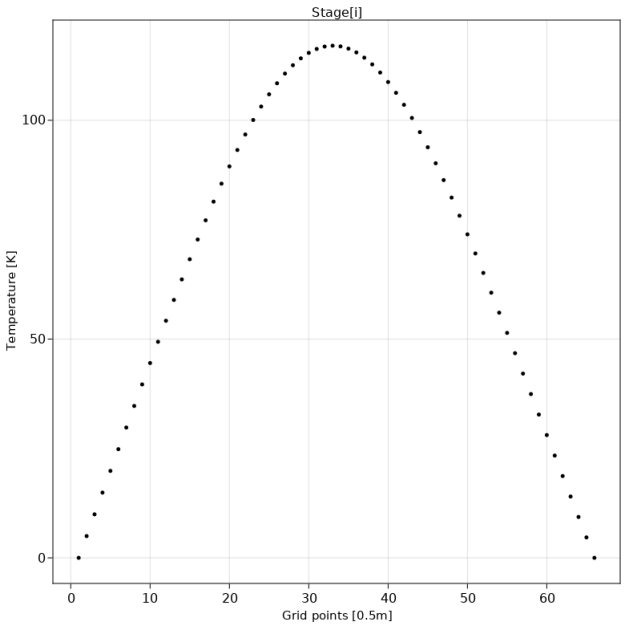

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.300 ns … 33.500 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.400 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.427 ns ±  0.537 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  1.3 ns         Histogram: frequency by time         1.7 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

Figure A.2: Two-dimensional computational costs on CPU

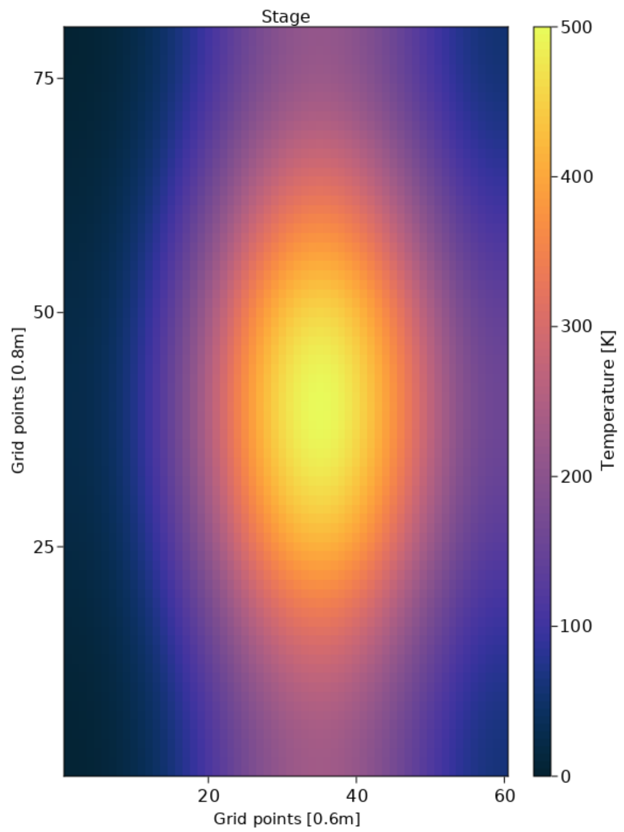
A.2 GPU Analysis




```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.300 ns … 21.300 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.500 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.553 ns ±  0.411 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  ▂▇▁                                                          ▁
  ▅▅▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁ ▁
  1.3 ns          Histogram: frequency by time          1.9 ns <
```

Figure A.3: One-dimensional computational costs on GPU




```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.200 ns … 29.900 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.300 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.398 ns ±  0.700 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  1.2 ns         Histogram: frequency by time        1.7 ns <
```

Figure A.4: Two-dimensional computational costs on GPU

Bibliography

[1] M. Chiappetta. "Nvidia geforce gtx 1050 and gtx 1050 ti review: Low power, low price pascal." (2016), [Online]. Available: https://hothardware.com/reviews/nvidia-geforce-gtx-1050--1050-ti-review.

[2] Nvidia. "Geforce gtx 1050 ti." (2022), [Online]. Available: https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1050-ti/specifications/.

[3] N. Sonawane and B. Nandwalkar, *r. Time Efficient Sentinel Data Mining using GPU*. 2015, https://www.ijert.org/time-efficient-sentinel-data-mining-using-gpu.

[4] NVIDA. "Cuda c++ programming guide, the programming guide to the cuda model and interface." (2022), [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[5] M. Jill Reese and S. Zaranek. "Gpu programming in matlab." (2015), [Online]. Available: https://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html.

[6] A. Rege. "Modern gpu architecture." (2015), [Online]. Available: http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf.

[7] D. C. B. Sobhan. "Modeling of flow and heat transfer in micro heat pipe." (2020), [Online]. Available: https://rpi.edu/dept/cct/public%20/TwoPhaseHeatTransferLab/research/mfht/mfht.html.

[8] E. Miersemann, *" Partial Differential Equations," chapter 1 "Introduction"*. 2012, https://www.math.uni-leipzig.de/~miersemann/pdebook.pdf.

[9] L. Evans, *Partial differential equations, Graduate Studies in Mathematics*. Americal Mathmatical society, May 1998.

[10] Simscale. "Boundary conditions." (2021), [Online]. Available: https://www.simscale.com/docs/simwiki/numerics-background/what-are-boundary-conditions/.

[11] J. Randall and LeVeque, *" Finite Difference Methods for Ordinary and Partial Differential Equations*. 2007, http://sgpwe.izt.uam.mx/files/users/uami/mlss/documentos/LeVequeRJ.pdf.

[12] North-Holland, *Handbook of Numerical Analysis, Numerical analysis part A*. 2022, https://www.ijert.org/time-efficient-sentinel-data-mining-using-gpu.

[13] ENCCS. "Solving heat equation with cuda." (2020), [Online]. Available: https://enccs.github.io/OpenACC-CUDA-beginners/2.02_cuda-heat-equation/.

[14] H. Emmons, *"The numerical solution of partial differential equations"*. 1944, https://www.ams.org/journals/qam/1944-02-03/S0033-569X-1944-10680-3/S0033-569X-1944-10680-3.pdf.

[15] W. E. Milne, *"T Numerical solution of differential equations"*. 1953, https://www.worldcat.org/title/numerical-solution-of-differential-equations/oclc/527661.

[16] R. S. U.M. Ascher S.J. Ruuth, *"Implicit-Explicit Runge-Kutta Methods for Time-Dependent Partial Differential Equations"*. 1997, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.1525&rep=rep1&type=pdf.

[17] C. Obbink-Huizer. "Implicit vs explicit finite element analysis: When to use which?" (2021), [Online]. Available: https://info.simuleon.com/blog/implicit-vs-explicit-finite-element-analysis.

[18] M. Bourne. "Euler's method - a numerical solution for differential equations." (2021), [Online]. Available: https://www.intmath.com/differential-equations/11-eulers-method-des.php.

[19] pctechguide. "Graphic card components." (2022), [Online]. Available: https://www.pctechguide.com/graphics-cards/graphic-card-components.

[20] . "Graphics processing unit architecture." (2021), [Online]. Available: http://ouhks260fminiproject.blogspot.com/2011/01/graphics-processing-unit-architecture.html.

[21] P. Fritzson. "Simplified schematic of nvidia gpu architecture, consisting of a set of streaming multiprocessors (sm)." (2012), [Online]. Available: https://www.researchgate.net/figure/Simplified-schematic-of-NVIDIA-GPU-architecture-consisting-of-a-set-of-Streaming_fig1_271848367.

[22] A. Lippert. "Nvidia gpu architecture for general purpose computing." (2012), [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[23] P. Gupta. "Cuda refresher: The cuda programming model." (2020), [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[24] github. "Cuda programming in julia." (2021), [Online]. Available: https://cuda.juliagpu.org/stable/tutorials/introduction/.

[25] T. Besard. "Cuda.jl 3.3." (2021), [Online]. Available: https://www.juliabloggers.com/cuda-jl-3-3/.

[26] C. Rackauckas. "Differentialequations.jl: Scientific machine learning (sciml) enabled simulation and estimation." (2022), [Online]. Available: https://diffeq.sciml.ai/stable/#DifferentialEquations.jl:-Scientific-Machine-Learning-(SciML)-Enabled-Simulation-and-Estimation.

[27] A. Koskela. "Ode solvers." (2015), [Online]. Available: https://diffeq.sciml.ai/stable/solvers/ode_solve/.

[28] ChrisRackauckas. "Diffeqgpu." (2015), [Online]. Available: https://github.com/SciML/DiffEqGPU.jl.

[29] T. Lienart. "Welcome to makie." (2022), [Online]. Available: https://makie.juliaplots.org/stable/.

[30] E. Kreyszig. "Advanced engineering mathematics." (2001), [Online]. Available: https://www.academia.edu/36591990/Kreyzig_12th_Advanced_Engineering_Mathematics_10th_Edition_pdf.

[31] F. A.Haight. "Mathematics in science and engineering." (1971), [Online]. Available: https://www.sciencedirect.com/science/article/pii/S007653920863012X.

[32] F. I. M.A. Demba. "A four-stage third-order symplectic explicit trigonometrically-fitted runge–kutta–nystrom¨ method for the numerical integration of oscillatory initial-value problems." (2016), [Online]. Available: https://www.researchgate.net/publication/311571860_A_four-stage_third-order_symplectic_explicit_trigonometrically-fitted_Runge-Kutta-Nystrom_method_for_the_numerical_integration_of_oscillatory_initial-value_problems.

[33] P. BOGACKI and L. F. SHAMPINE. "An efficient runge-kutta (4,5) pair." (1996), [Online]. Available: https://core.ac.uk/download/pdf/81941641.pdf.

[34] D. Boffi. "Computers mathematics with applications." (2022), [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122111004706.

[35] D. I. K. Sigal Gottlieb and Chi-Wang. "Strong stability preserving runge-kutta and multistep time discretizations." (2011), [Online]. Available: https://books.google.de/books?id=MHmAINTBIkQC&printsec=frontcover&dq=explicit+strong+stability+preserving&hl=ar&sa=X&redir_esc=y#v=onepage&q=explicit%20strong%20stability%20preserving&f=false.

[36] E. Hairer. "Solving ordinary differential equations i nonstiff problems." (2008), [Online]. Available: https://www.google.de/books/edition/Solving_Ordinary_Differential_Equations/cfZDAAAAQBAJ?hl=en&gbpv=0.