Article

GPU Accelerating Algorithms for Three-Layered Heat Conduction Simulations

Nicolás Murúa [1], Aníbal Coronel [2,*], Alex Tello [3], Stefan Berres [4,5] and Fernando Huancas [6]

[1] Departamento de Ciencias de la Computación y Tecnologías de la Información, Facultad de Ciencias Empresariales, Universidad del Bío-Bío, Campus Fernando May, Chillán 3780000, Chile; nicolas.murua1701@alumnos.ubiobio.cl
[2] Departamento de Ciencias Básicas, Centro de Ciencias Exactas UBB (CCE-UBB), Facultad de Ciencias, Universidad del Bío-Bío, Campus Fernando May, Chillán 3780000, Chile
[3] Departamento de Matemáticas, Facultad de Ciencias Básicas, Universidad de Antofagasta, Antofagasta 1270300, Chile; alex.tello@uantof.cl
[4] Núcleo de Investigación en Bioproductos y Materiales Avanzados (BioMA), Universidad Católica de Temuco, Temuco 4780002, Chile; stefan.berres@gmail.com
[5] Integrata-Stiftung für Humane Nutzung der Informationstechnologie, Vor dem Kreuzberg 28, 72070 Tübingen, Germany
[6] Departamento de Matemática, Facultad de Ciencias Naturales, Matemáticas y del Medio Ambiente, Universidad Tecnológica Metropolitana, Las Palmeras 3360, Ñuñoa, Santiago 7750000, Chile; fhuancas@utem.cl
* Correspondence: acoronel@ubiobio.cl

Abstract: In this paper, we consider the finite difference approximation for a one-dimensional mathematical model of heat conduction in a three-layered solid with interfacial conditions for temperature and heat flux between the layers. The finite difference scheme is unconditionally stable, convergent, and equivalent to the solution of two linear algebraic systems. We evaluate various methods for solving the involved linear systems by analyzing direct and iterative solvers, including GPU-accelerated approaches using CuPy and PyCUDA. We evaluate performance and scalability and contribute to advancing computational techniques for modeling complex physical processes accurately and efficiently.

Keywords: sparse linear systems; finite difference method; heat transfer; GPU acceleration; high-performance computing; parallel processing; computational efficiency

MSC: 65M06; 65M12; 76A20

Citation: Murúa, N.; Coronel, A.; Tello, A.; Berres, S.; Huancas, F. GPU Accelerating Algorithms for Three-Layered Heat Conduction Simulations. *Mathematics* **2024**, *12*, 3503. https://doi.org/10.3390/math12223503

Academic Editors: Hao Jiang and Zhe Quan

Received: 3 October 2024
Revised: 4 November 2024
Accepted: 6 November 2024
Published: 9 November 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

1.1. Application Context

In recent years, research and development efforts have increasingly focused on advancing energy sources, with particular attention to reducing greenhouse gas emissions, managing fuel costs, optimizing energy demand, expanding renewable energy production, and designing materials that enhance the efficiency of energy production and distribution. A significant objective is also to replace fossil fuels with more sustainable alternatives [1].

Increasing demands for energy-efficient materials have spurred interest in multilayered materials, particularly for applications requiring precise thermal management, such as insulation, electronics, and aerospace engineering. Multilayered structures provide tailored thermal properties through variations in composition and thickness, making them valuable for optimizing heat flow and thermal resistance. Indeed, it is known that the design of multilayered materials is a relevant area of research as they are present in numerous applications such as metal, alloy production, casting, heating, and building construction [2–8]. Therefore, studying heat conduction in multilayered materials is crucial for improving energy control and optimizing usage [9–14].

1.2. Core Scientific Question

Accurately simulating heat conduction in these materials poses a significant challenge due to the interactions between layers with diverse properties. This study addresses these challenges by developing GPU-accelerated numerical methods to improve the efficiency and accuracy of multilayered material simulations. More precisely, the core scientific question of this contribution can be stated as follows:

> How can GPU-accelerated solvers enhance the computational efficiency and accuracy of simulating heat conduction in multi-layered solids, particularly when solving large, sparse linear systems generated by finite difference methods, and which solver strategies (direct versus iterative) are optimal for managing the stability and condition number challenges inherent in these models?

1.3. Numerical Modelling

The heat transfer in a multilayered solid is of practical importance in several devices such as batteries, semiconductors, thermopane windows, and nuclear reactors [15–17]. In a broad sense, the mathematical analysis of the mathematical models is developed by generalizing the standard techniques for classical heat conduction, such as separation of variables, adjoint method, semigroup theory, and finite integral transform method. Meanwhile, the numerical solution can be obtained by application of finite difference, finite volume, or finite element methods [18–20].

In this paper, we consider that the heat transmission is modeled by the dual-phase-lagging equation with appropriate interface and boundary conditions, which is obtained by following the recent approach studied in [21]. We discretize the model by using a finite difference method. In a broad sense, the finite difference method discretizes partial differential equations into a system of algebraic equations, typically resulting in large, sparse, and structured linear systems. These systems are frequently solved iteratively to obtain numerical solutions that approximate the behavior of the underlying continuous physical phenomena. The accuracy, stability, and convergence of the numerical solution heavily depend on the effectiveness of the linear system solver employed. Particularly, for the finite difference discretization of the three-layer solid model of heat conduction, we consider the finite difference scheme introduced in [21], which can be summarized as the solution of two linear algebraic systems.

1.4. Computational Simulation

On the other hand, we know that linear systems of equations play a fundamental role in numerous scientific and engineering applications, particularly in computational methods such as the finite difference method [22–24]. These systems arise in various fields, from fluid dynamics and heat transfer to structural analysis and electromagnetics. In the context of the finite difference method, which is widely employed for solving partial differential equations numerically, the efficient and accurate solution of linear systems is of paramount importance. Particularly, in this paper, we explore and evaluate various methods for solving the linear systems arising from the finite difference method for discretization of the three-layer heat-transfer model. We investigate different solution techniques, including direct and iterative methods, and considering their applicability, efficiency, and numerical stability, this report seeks to provide insights into the optimal strategies for addressing linear systems encountered in finite difference simulations. Through a comprehensive analysis, we aim to contribute to advancing computational techniques for modeling and simulating complex physical processes accurately and efficiently. Moreover, we advance the study of parallel programming methodologies [25].

1.5. Main Contributions

The main contributions of the paper are the study of the solution of the linear systems associated with the finite difference methods. We remark that the solvers are not only assessed for their performance in solving linear systems, but also tested within the

GPU-accelerated computing framework. Leveraging the capabilities of modern GPUs can significantly enhance the computational efficiency of numerical simulations, particularly for large-scale problems encountered in finite difference simulations. To harness GPU acceleration, the calculations are carried out using CuPy [26], a GPU-accelerated library that provides a NumPy-like interface for array operations on CUDA-enabled GPUs. Furthermore, the finite difference computations are performed using PyCUDA [27], a Python (Version 3.10.12) library that facilitates GPU programming with CUDA, where the kernels were implemented from scratch to exploit the parallel processing power of GPUs efficiently. By utilizing these existing GPU-accelerated libraries alongside custom CUDA kernels developed using PyCUDA, we aim to evaluate the effectiveness of GPU use in accelerating the solution of linear systems within the finite difference method. Through rigorous testing and benchmarking, we seek to identify the most efficient and scalable solver strategies that leverage the computational capabilities of modern GPU architectures.

The paper is organized as follows. In Section 2, we present the mathematical model and the finite difference scheme. In Section 3, we present the GPU algorithm. In Section 4, we present the results for the performance of the proposed algorithm. Finally, in Section 5, we present some conclusions and future work.

2. Mathematical Model and Finite Difference Scheme

In this section, we discuss the mathematical model for heat conduction on the three-layered solid and the discretization using the finite difference method. We begin with a discussion of the standard theory of heat conduction and introduce the phase-lag heat equation in a single-layered system, and then we extend to the case of a three-layered solid by incorporating the appropriate conditions on the interface. Meanwhile, in principle, discretization can be developed by finite difference, finite volume, or other techniques that are used for parabolic equations. Here, we consider the finite difference method as it has been effectively used to address the approximation of the model in [21]. The method is unconditionally convergent, but, in practical implementations, it has the disadvantage of high condition numbers of the involved matrices when the spatial discretizations are refined. We remark that the theoretical calculus of condition numbers is complex, but will be evidenced in our numerical simulations.

2.1. Heat Conduction Mathematical Model

In this article, we consider three-layered solids, as shown in Figure 1, see also Table 1 for notations and terminology. We assume that the solids of the three layers satisfy the physical restrictions, such that the heat conduction can be modeled by the dual-phase-lagging approach or the generalized Fourier law introduced in [13] (see also [14,28,29]).

Table 1. Physical and geometrical notation for the three-layered solid given in Figure 1.

Notation	Definition
Geometrical	
$\overline{W}_\ell$	width of ℓth-layer
$L_0 = 0$	left boundary
$L_1 = \overline{W}_1$	interface 1
$L_2 = \overline{W}_1 + \overline{W}_2$	interface 2
$L_3 = L = \overline{W}_1 + \overline{W}_2 + \overline{W}_3$	right boundary
$\mathcal{I}_\ell =]L_{\ell-1}, L_\ell[$	interval denoting the ℓth-layer
$\mathcal{I}^{lay} = \cup_{\ell=1}^{3} \mathcal{I}_\ell, \ \ \overline{\mathcal{I}^{lay}} = [L_0, L_3]$	space domain
$[0, T]$	time domain
$Q_T^{lay} = \cup_{\ell=1}^{3} Q_{\ell,T}, \ \ Q_{\ell,T} = \mathcal{I}_\ell \times [0, T]$	space–time domain

Table 1. *Cont.*

Notation	Definition
Physical	
C^ℓ	the heat capacitance of ℓth-layer
τ_q^ℓ	heat flux phase lags of ℓth-layer
τ_T^ℓ	temperature gradient phase lags of ℓth-layer
k_ℓ	thermal conductivity of ℓth-layer
α_1, α_2	some proportionality constants
$\overline{K}_1, \overline{K}_2$	Knudsen numbers
$f_\ell(x,t)$	heat source function of ℓth-layer
$\psi_1(x)$	initial distribution of the temperature
$\psi_2(x)$	initial distribution of the temporal derivative of temperature
$\varphi_1(t)$	temperature flux at the left boundary of the solid
$\varphi_2(t)$	temperature flux at the right boundary of the solid

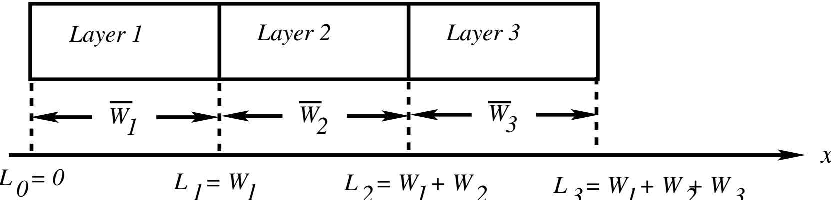


Figure 1. Visual representation of the three-layered solid.

Let us recall some terminology and notation of heat conduction. The classical model for heat conduction in a region $\Omega \subset \mathbb{R}^d$ ($d = 1, 2, 3$) is given by the currently known heat equation:

$$\frac{\partial T}{\partial t}(x,t) = \alpha \Delta T(x,t) + S(x,t), \tag{1}$$

where t denotes the time, $x \in \Omega$ is the space position, T is the temperature, $\alpha > 0$ is the thermal diffusivity of the medium, and S is the volumetric heat generation or source of heat. Equation (1) is deduced by developing the differential form of the temperature balance and the Fourier law for heat flux:

$$C\frac{\partial T}{\partial t}(x,t) = -\mathrm{div}(\mathbf{q}(x,t)) + \overline{Q}(x,t), \tag{2}$$

$$\mathbf{q}(x,t) = -k\nabla T(x,t), \tag{3}$$

where C denotes the heat capacity of the material, $\mathbf{q}$ is the heat flux, $\overline{Q}$ is the volumetric heat generation, and k is the thermal conductivity of the material. We deduce (1), from (3) in (2) with $\alpha = k/C$ and $S = \overline{Q}/C$. It is known that the model (1) has disadvantages; for instance, it is not adequate for the mathematical modelling the heat transport at very high frequencies and short wavelengths. Then, some improvements of (1) are proposed, for instance the model

$$\tau\frac{\partial^2 T}{\partial t^2}(x,t) + \frac{\partial T}{\partial t}(x,t) = \frac{k}{C}\Delta T(x,t) + \frac{1}{C}\left(\overline{Q}(x,t) + \tau\frac{\partial \overline{Q}}{\partial t}(x,t)\right), \quad \tau > 0, \tag{4}$$

is introduced by Cattaneo by considering the heat flux is modeled by

$$\tau\frac{\partial \mathbf{q}}{\partial t}(x,t) = -\Big(\mathbf{q}(x,t) + k\nabla T(x,t)\Big), \tag{5}$$

where τ is a positive constant denoting a relaxation parameter. If the relaxation parameter vanishes, $\tau \to 0$, the model (4) converges to (1). A more recent extension considers that the heat flux is modeled by

$$\mathbf{q}(x, t + \tau_q) = -k\nabla T(x, t + \tau_T), \tag{6}$$

where τ_T and τ_q are the phase lags for the temperature gradient and heat flux, respectively [13]. If $\Omega \subset \mathbb{R}$, a Taylor expansion in (6) implies that

$$q(x,t) + \tau_q \frac{\partial q}{\partial x}(x,t) = -k\left[\frac{\partial T}{\partial x}(x,t) + \tau_T \frac{\partial^2 T}{\partial t \partial x}(x,t)\right]. \tag{7}$$

From (2) and (7), we obtain the model for heat conduction given by

$$C\left(\frac{\partial T}{\partial t} + \tau_q \frac{\partial^2 T}{\partial t \partial x}\right) = k\left(\frac{\partial^2 T}{\partial x^2} + \tau_T \frac{\partial^3 T}{\partial t \partial^2 x}\right) + \overline{Q}(x,t) + \tau_q \frac{\partial \overline{Q}}{\partial x}(x,t), \tag{8}$$

which is known as the dual-phase-lagging heat conduction equation. If the phase lags parameter vanishes, i.e., $(\tau_q, \tau_T) \to (0,0)$, the model (8) is reduced to (1).

Let us consider a three-layered solid of the schematic form given in (1) and assume that the heat conduction on each solid is given by (8). Then, heat conduction in a three-layered solid is modeled by

$$C(x)\left(\frac{\partial u}{\partial t} + \tau_q(x) \frac{\partial^2 u}{\partial t^2}\right) = k(x)\left(\frac{\partial^2 u}{\partial x^2} + \tau_T(x) \frac{\partial^3 u}{\partial t \partial x^2}\right) + f(x,t), \quad (x,t) \in Q_T^{lay}, \tag{9}$$

where u is the temperature, $C(x)$ is the heat capacity, $k(x)$ is the thermal conductivity, $\tau_q(x)$ is the phase lag of the heat flux, $\tau_T(x)$ is the phase lag of the temperature gradient, and $f(x,t)$ is the volumetric heat generation [21]. The coefficients and source functions are piecewise functions of the following form

$$\begin{aligned}
C(x) &= \sum_{\ell=1}^{3} C_\ell \mathbb{1}_{[L_{\ell-1}, L_\ell)}(x), & k(x) &= \sum_{\ell=1}^{3} k_\ell \mathbb{1}_{[L_{\ell-1}, L_\ell)}(x), \\
\tau_q(x) &= \sum_{\ell=1}^{3} \tau_q^{(\ell)} \mathbb{1}_{[L_{\ell-1}, L_\ell)}(x), & \tau_T(x) &= \textstyle\sum_{\ell=1}^{3} \tau_T^{(\ell)} \mathbb{1}_{[L_{\ell-1}, L_\ell)}(x), \\
f(x,t) &= \textstyle\sum_{\ell=1}^{3} f_\ell(x,t) \mathbb{1}_{[L_{\ell-1}, L_\ell)}(x),
\end{aligned}$$

where $\mathbb{1}_A$ is the indicator function, defined as $\mathbb{1}_A(x) = 1$ if $x \in A$ and $\mathbb{1}_A(x) = 0$ otherwise. The governing Equation (9) is supplemented with initial, boundary, and interface conditions, as specified below. The initial conditions are defined by

$$u(x,0) = \psi_1(x) \qquad \frac{\partial u}{\partial t}(x,0) = \psi_2(x), \quad x \in \overline{\mathcal{I}^{lay}}. \tag{10}$$

For modelling the boundary conditions, we consider zero-flow at the two boundaries:

$$\left(-\alpha_1 \overline{K}_1 \frac{\partial u}{\partial x} + u\right)(L_0, t) = \varphi_1(t), \qquad \left(\alpha_2 \overline{K}_2 \frac{\partial u}{\partial x} + u\right)(L_3, t) = \varphi_2(t), \quad t \in [0,T], \tag{11}$$

Related to the interface conditions, we consider that the temperature and heat flux at the interface L_1 and L_2 are given by

$$[\![u(x,t)]\!] = 0, \qquad \left[\!\!\left[k(x)\left(\frac{\partial u}{\partial x} + \tau_T(x) \frac{\partial^2 u}{\partial x \partial t}\right)(x,t)\right]\!\!\right] = 0, \quad (x,t) \in I_T^{int}, \tag{12}$$

where $[\![\cdot]\!]$ denotes the jump across the interface and $I_T^{int} := \{L_1, L_2\} \times [0,T]$. This model captures the heat conduction behavior in a multi-layered medium, accounting for the

dual-phase-lagging effect and ensuring continuity of temperature and heat flux across layer interfaces.

2.2. Finite Difference Method to Approximate (9)–(12)

The discretization of the mathematical model (9)–(12) is extensively discussed in [21] and can be summarized as follows. Let us consider the notation in Table 1. We begin by discretizating of space and time domains by selecting $m_1, m_2, m_3, N \in \mathbb{N}$, and considering that the ℓ-th layer $\mathcal{I}_\ell$ and $[0,T]$ are divided into m_ℓ and N parts of sizes Δx_ℓ and Δt, respectively. Moreover, we introduce the notation

$$\begin{aligned}
&M_0 = 0 \text{ and } M_\ell = \textstyle\sum_{i=1}^{\ell} m_i \text{ for } \ell \in \{1,2,3\}, \quad \Delta W_\ell = (L_\ell - L_{\ell-1})/m_\ell, \\
&x_j = L_{\ell-1} + (j - M_{\ell-1})\Delta W_\ell \text{ for } (\ell, j) \in \{1,2,3\} \times \{M_{\ell-1}, \dots, M_\ell\}, \\
&\Delta x_j = \Delta W_\ell \text{ for } j \in \{M_{\ell-1}, \dots, M_\ell\}, \quad \mathcal{I}_{\Delta x}^{int} = \{x_j \,:\, j = M_1, M_2\}, \\
&\mathcal{I}_{\ell,\Delta x} = \{x_j \,:\, j = M_{\ell-1}+1, \dots, M_\ell - 1\}, \quad \mathcal{I}_{\Delta x}^{lay} = \cup_{\ell=1}^{d} \mathcal{I}_{\ell,\Delta x}, \\
&\partial\mathcal{I}_{\Delta x}^{lay} = \{x_j \,:\, j = M_0, M_3\}, \quad \overline{\mathcal{I}}_{\Delta x} = \mathcal{I}_{\Delta x}^{lay} \cup \mathcal{I}_{\Delta x}^{int} \cup \partial\mathcal{I}_{\Delta x}^{lay}, \\
&\Delta t = T/N, \quad t_n = n\Delta t \text{ for } n = 0, \dots, N, \quad \mathcal{T}_{\Delta t} = \{t_n \,:\, n = 0, \dots, N\}, \\
&Q_{\Delta x,\Delta t} = \overline{\mathcal{I}}_{\Delta x} \times \mathcal{T}_{\Delta t}, \mathbb{I} = \{M_0, \dots, M_3\}, \mathbb{I}^{int} = \{M_1, M_2\}, \\
&\partial\mathbb{I} = \{M_0, M_3\}, \mathbb{I}^{lay} = \mathbb{I} - (\mathbb{I}^{int} \cup \partial\mathbb{I}).
\end{aligned}$$

We remark that the symbol M_ℓ is defined to identify the index for interfaces and boundaries; for instance, for $j = M_0$, we have the left boundary. In order to discretize the equations, we denote the approximation of the temperature as follows:

$$\mathcal{U}_{\Delta x,\Delta t} = \Big\{ \mathbb{U} = (\mathbf{u}^0, \dots, \mathbf{u}^N) \in \mathbb{R}^{M_d+1} \times \mathbb{R}^{N+1} \quad : \quad \mathbf{u}^n = (u_0^n, \dots, u_{M_3}^n) \Big\}.$$

where u_j^n is defined by $u_j^n = u(x_j, t_n)$ for $(j,n) \in \mathbb{I} \times \{0, \dots, N\}$. In addition, we consider that the following extended notation for initial, boundary, and interface conditions

$$\begin{aligned}
&\boldsymbol{\varphi}_k = (\varphi_k^0, \dots, \varphi_k^N), \quad \varphi_k^n = \varphi_k(t_n), \quad n \in \{0,1,\dots,N\}, \quad k \in \{0,1\}; \\
&\boldsymbol{\delta\varphi}_k = (\varphi_k'^0, \dots, \varphi_k'^N), \quad \varphi_k'^n = \varphi_k'(t_n), \quad n \in \{0,1,\dots,N\}, \quad k \in \{0,1\}; \\
&\boldsymbol{\phi}_1 = \boldsymbol{\varphi}_1 + \tau_T(x_{M_0})\boldsymbol{\delta\varphi}_1; \quad \boldsymbol{\phi}_2 = \boldsymbol{\varphi}_2 + \tau_T(x_{M_3})\boldsymbol{\delta\varphi}_2; \\
&\mathcal{L}_j^C = \frac{C(x_j)}{(\Delta t)^2}\Big(\Delta t + 2\tau_q(x_j)\Big), \quad \mathcal{L}_j^L = 2C(x_j)\frac{\tau_q(x_j)}{(\Delta t)^2}, \quad j \in \mathbb{I}; \\
&\Psi_j^R = 1 + \frac{\tau_T(x_j)}{\Delta t}, \quad \Psi_j^L = -\frac{\tau_T(x_j)}{\Delta t}, \quad j \in \mathbb{I}; \\
&\mu_j = \frac{k(x_j)}{(\Delta x_j)^2}, \quad \Psi_j^\pm = \left(\frac{1}{2} \pm \frac{\tau_T(x_j)}{\Delta t}\right), \quad j \in \mathbb{I}; \\
&\mathcal{Z}_j^L = \frac{C(x_j)}{2(\Delta t)^2}\Big(-\Delta t + 2\tau_q(x_j)\Big), \; \mathcal{Z}_j^U = \frac{C(x_j)}{2(\Delta t)^2}\Big(\Delta t + 2\tau_q(x_j)\Big), \\
&\mathcal{Z}_j^C = \mathcal{Z}_j^L + \mathcal{Z}_j^U, \quad j \in \mathbb{I}.
\end{aligned}$$

Hence, the numerical approximation of (9)–(12) is given by the following scheme

$$\hat{A}u^{n+1} = \hat{B}u^n + \hat{s}, \qquad \text{for} \quad n = 0, \tag{13}$$

$$Au^{n+1} = Bu^n + Cu^{n-1} + s^n, \qquad \text{for} \quad n = 1, \dots, N, \tag{14}$$

where $\hat{A}$, $\hat{B}$, A, B, and C are tridiagonal matrices and $\hat{s}$ and s^n are vectors defined in Tables 2 and 3. This makes the method multi-step, allowing for higher accuracy and enabling the capture of dynamics introduced by the phase lags. An in-depth description of the model and the numerical scheme can be found in the formulation from [21].

Table 2. Entries of the tridiagonal matrices $\hat{\mathbb{A}}$, $\hat{\mathbb{B}}$ and the vector $\mathbf{s}^0$ defining the system (13).

		under-diagonal $i = j+1$	diagonal $i = j$	upper-diagonal $i = j-1$
$\hat{a}_{i,j}$	$j = M_0$		$\mathcal{L}^C_{M_0} + 2\mu_{M_0}\left(1 + \frac{\Delta x_{M_0}}{\alpha_1 \overline{K}_1}\right)\Psi^R_{M_0}$	$-2\mu_{M_0}\Psi^R_{M_0}$
	$j \in \mathbb{I}^{lay}$	$-\mu_j \Psi^R_j$	$\mathcal{L}^C_j + 2\mu_j \Psi^R_j$	$-\mu_j \Psi^R_j$
	$j \in \mathbb{I}^{int}$	$-2\Delta x_{j-1}\mu_{j-1}\Psi^R_{j-1}$	$\Delta x_{j-1}\left(\mathcal{L}^C_j + 2\mu_{j-1}\Psi^R_j\right) + \Delta x_j\left(\mathcal{L}^C_j + 2\mu_j \Psi^R_j\right)$	$-2\Delta x_j \mu_j \Psi^R_j$
	$j = M_3$	$2\mu_{M_3-1}\Psi^R_{M_3-1}$	$\mathcal{L}^C_{M_3} + 2\mu_{M_3-1}\left(1 + \frac{\Delta x_{M_3-1}}{\alpha_2 \overline{K}_2}\right)\Psi^R_{M_3-1}$	
		under-diagonal $i = j+1$	diagonal $i = j$	upper-diagonal $i = j-1$
$\hat{b}_{i,j}$	$j = M_0$		$\mathcal{L}^C_{M_0} - 2\mu_{M_0}\left(1 + \frac{\Delta x_{M_0}}{\alpha_1 \overline{K}_1}\right)\Psi^L_{M_0}$	$2\mu_{M_0}\Psi^L_{M_0}$
	$j \in \mathbb{I}^{lay}$	$\mu_j \Psi^L_j$	$\mathcal{L}^C_j - 2\mu_j \Psi^L_j$	$\mu_j \Psi^L_j$
	$j \in \mathbb{I}^{int}$	$2\Delta x_{j-1}\mu_{j-1}\Psi^L_{j-1}$	$\Delta x_{j-1}\left(\mathcal{L}^C_j - 2\mu_{j-1}\Psi^L_{j-1}\right) + \Delta x_j\left(\mathcal{L}^C_j - 2\mu_j \Psi^L_j\right)$	$2\Delta x_j \mu_j \Psi^L_j$
	$j = M_3$	$2\mu_{M_3-1}\Psi^L_{M_3-1}$	$\mathcal{L}^C_{M_3} - 2\mu_{M_3-1}\left(1 + \frac{\Delta x_{M_3-1}}{\alpha_2 \overline{K}_2}\right)\Psi^L_{M_3-1}$	
s^0_j	$j = M_0$	$\mathcal{L}^U_{M_0}\Delta t \psi_2(x_{M_0}) + 2\mu_{M_0}\frac{\Delta x_{M_0}}{\alpha_1 \overline{K}_1}\phi^1_1 + f^1_{M_0}$		
	$j \in \mathbb{I}^{lay}$	$\mathcal{L}^U_j \Delta t \psi_2(x_j) + f^1_j$		
	$j \in \mathbb{I}^{int}$	$\left(\Delta x_{j-1} + \Delta x_j\right)\mathcal{L}^L_j \Delta t \psi_2(x_j) + \Delta x_{j-1} f^1_{j-1} + \Delta x_j f^1_j$		
	$j = M_3$	$\mathcal{L}^U_{M_3}\Delta t \psi_2(x_{M_3}) + 2\mu_{M_3-1}\frac{\Delta x_{M_3-1}}{\alpha_2 \overline{K}_2}\phi^1_2 + f^1_{M_3-1}$		

Table 3. Entries of the tridiagonal matrices $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$ and the vector $\mathbf{s}^n$ defining the system (14).

		under-diagonal $i = j+1$	diagonal $i = j$	upper-diagonal $i = j-1$
$a_{i,j}$	$j = M_0$		$\mathcal{Z}^U_{M_0} + \mu_{M_0}\left(1 + \frac{\Delta x_{M_0}}{\alpha_1 \overline{K}_1}\right)\Psi^+_{M_0}$	$-\mu_{M_0}\Psi^+_{M_0}$
	$j \in \mathbb{I}^{lay}$	$-\frac{1}{2}\mu_j \Psi^+_j$	$\mathcal{Z}^U_j + \mu_j \Psi^+_j$	$-\frac{1}{2}\mu_j \Psi^+_j$
	$j \in \mathbb{I}^{int}$	$-\Delta x_{j-1}\mu_{j-1}\Psi^+_{j-1}$	$\Delta x_{j-1}\left(\mathcal{Z}^U_j + \mu_{j-1}\Psi^+_{j-1}\right) + \Delta x_j\left(\mathcal{Z}^U_j + \mu_j \Psi^+_j\right)$	$-\Delta x_j \mu_j \Psi^+_j$
	$j = M_3$	$-\mu_{M_3-1}\Psi^+_{M_3-1}$	$\mathcal{Z}^U_{M_3} + \mu_{M_3-1}\left(1 + \frac{\Delta x_{M_3-1}}{\alpha_2 \overline{K}_2}\right)\Psi^+_{M_3-1}$	
		under-diagonal $i = j+1$	diagonal $i = j$	upper-diagonal $i = j-1$
$b_{i,j}$	$j = M_0$		$\mathcal{Z}^C_{M_0} - \mu_{M_0}\left(1 + \frac{\Delta x_{M_0}}{\alpha_1 \overline{K}_1}\right)$	μ_{M_0}
	$j \in \mathbb{I}^{lay}$	$\frac{1}{2}\mu_j$	$\mathcal{Z}^C_j - \mu_j$	$\frac{1}{2}\mu_j$
	$j \in \mathbb{I}^{int}$	$\Delta x_{j-1}\mu_{j-1}$	$\Delta x_{j-1}\left(\mathcal{Z}^C_j - \mu_{j-1}\right) + \Delta x_j\left(\mathcal{Z}^C_j - \mu_j\right)$	$\Delta x_j \mu_j$
	$j = M_3$	μ_{M_3-1}	$\mathcal{Z}^C_{M_3} - \mu_{M_3-1}\left(1 + \frac{\Delta x_{M_3-1}}{\alpha_2 \overline{K}_2}\right)$	
		under-diagonal $i = j+1$	diagonal $i = j$	upper-diagonal $i = j-1$
$c_{i,j}$	$j = M_0$		$-\left[\mathcal{Z}^L_{M_0} + \mu_{M_0}\left(1 - \frac{\Delta x_{M_0}}{\alpha_1 \overline{K}_1}\right)\Psi^-_{M_0}\right]$	$\mu_{M_0}\Psi^-_{M_0}$
	$j \in \mathbb{I}^{lay}$	$\frac{1}{2}\mu_j \Psi^-_j$	$-\left[\mathcal{Z}^L_j + \mu_j \Psi^-_j\right]$	$\frac{1}{2}\mu_j \Psi^-_j$
	$j \in \mathbb{I}^{int}$	$\Delta x_{j-1}\mu_{j-1}\Psi^-_{j-1}$	$-\left[\Delta x_{j-1}\left(\mathcal{Z}^L_j + \mu_{j-1}\Psi^-_{j-1}\right) + \Delta x_j\left(\mathcal{Z}^L_j + \mu_j \Psi^-_j\right)\right]$	$\Delta x_j \mu_j \Psi^-_j$
	$j = M_3$	$\mu_{M_3}\Psi^-_{M_3-1}$	$-\left[\mathcal{Z}^L_{M_3} + \mu_{M_3-1}\left(1 + \frac{\Delta x_{M_3-1}}{\alpha_2 \overline{K}_2}\right)\Psi^-_{M_3}\right]$	

Table 3. *Cont.*

s_j^n	$j = M_0$	$\mu_{M_0} \dfrac{\Delta x_{M_0}}{2\alpha_1 \overline{K}_1} \left(\phi_1^{n-1} + 2\phi_1^n + \phi_1^{n+1}\right) + \dfrac{1}{4}\left(f_{M_0}^{n-1} + 2f_{M_0}^n + f_{M_0}^{n+1}\right)$
	$j \in \mathbb{I}^{lay}$	$\dfrac{1}{4}(f_j^{n-1} + 2f_j^n + f_j^{n+1})$
	$j \in \mathbb{I}^{int}$	$\dfrac{\Delta x_{j-1}}{4}(f_{j-1}^{n-1} + 2f_{j-1}^n + f_{j-1}^{n+1}) + \dfrac{\Delta x_j}{4}(f_j^{n-1} + 2f_j^n + f_j^{n+1})$
	$j = M_3$	$\mu_{M_3-1} \dfrac{\Delta x_{M_3-1}}{2\alpha_2 \overline{K}_2} \left(\phi_2^{n-1} + 2\phi_2^n + \phi_2^{n+1}\right) + \dfrac{1}{4}(f_{M_3}^{n-1} + 2f_{M_3}^n + f_{M_3}^{n+1})$

3. The GPU Accelerating Algorithm

The numerical computation considered in this paper consists of three main sections, leveraging both CPU and GPU processing for optimal performance. To be specific, we consider five steps, which are summarized in Table 4 and described in continuation:

Steps 1 and 2: CPU Pre-Computation. Initially, Steps 1 and 2, are computed on the CPU (host) due to their minimal computational overhead. These steps involve the definition of input data and the discretization of space and time. The resulting values are then allocated in the memory on the GPU (device) to prepare for further processing.

Steps 3 and 4: GPU Pre-Computation. Steps 3 and 4 are executed on the GPU using a set of CUDA kernels invoked with PyCUDA. This section involves memory allocation on the GPU and the execution of CUDA kernels to perform computations efficiently. By keeping the data on the device, the linear solver can access it easily, reducing overheads associated with transferring data between the CPU and GPU. At the conclusion of this step, unnecessary data are deallocated, retaining only the matrices and arrays required to solve the linear system and compute the solution vector u.

Step 5: Compute the evolution of the system with GPU. Step 5 involves the use of a CPU function that calls the GPU functions implemented with CuPy to solve the systems using various numerical methods. Despite the GPU acceleration for the operations, a constant call to independent and sequential kernels is made to compute the right-hand side of the equation and solve the linear system for each iteration $n + 1$ after completing iteration n.

Table 4. Description of CPU/GPU usage at different steps.

Step	Step Description	CPU/GPU Usage
1	Definition of Input Data	CPU Pre-Computation
2	Discretization of Space and Time	CPU Pre-Computation
3	Evaluation of Functions on the Mesh	GPU Pre-Computation
4	Calculation of Matrices and Vectors	GPU Pre-Computation
5	Discretization of Equations	Compute evolution of the system with GPU

In Figure 2, we provide a general flowchart based on those steps. It does not show the complete interaction between the CPU and the GPU during the process of running the program, but it shows a simplification that provides insight into the locations where each workload is executed. Furthermore, in Figure 3, we show the complete flowchart without the divisions on the main step. Mainly, for each process, the memory allocation and deallocation on the GPU is controlled from the CPU.

In this paper, the kernels for computing Steps 3 and 4 were programmed on GPU with CUDA and executed with PyCUDA. The functions are straight forward to parallelize, given that all of them consist of computing values of an array or matrix without dependencies between the data. We can simply divide the work on as many threads as the size of the output and structure the kernel, so that each thread computes only its corresponding position. An example of a function implemented sequentially can be found in Algorithm 1, while Algorithm 2 shows the kernel that the GPU will execute in parallel, and the procedure for the CPU to invoke the kernel. The matrix $\mathcal{F}$ corresponds to the discretization of $f(x, t)$,

the source heat generation in (9). Meanwhile, Algorithm 2 computes the matrix $\mathcal{F}$ by organizing the threads on a 2D grid; when each thread starts, it identifies its position and computes the associated value.

Similar sequential and parallel algorithms are designed for the computation of C, k, τ_q and τ_T. More precisely, all functions in Step 3 and 4 are parallelized with this logic, the only difference exists if the output is a 1D-array or a 2D-Matrix, which would require defining the kernel as 1D or 2D grid. Other than this, the approach to GPU acceleration remains largely the same.

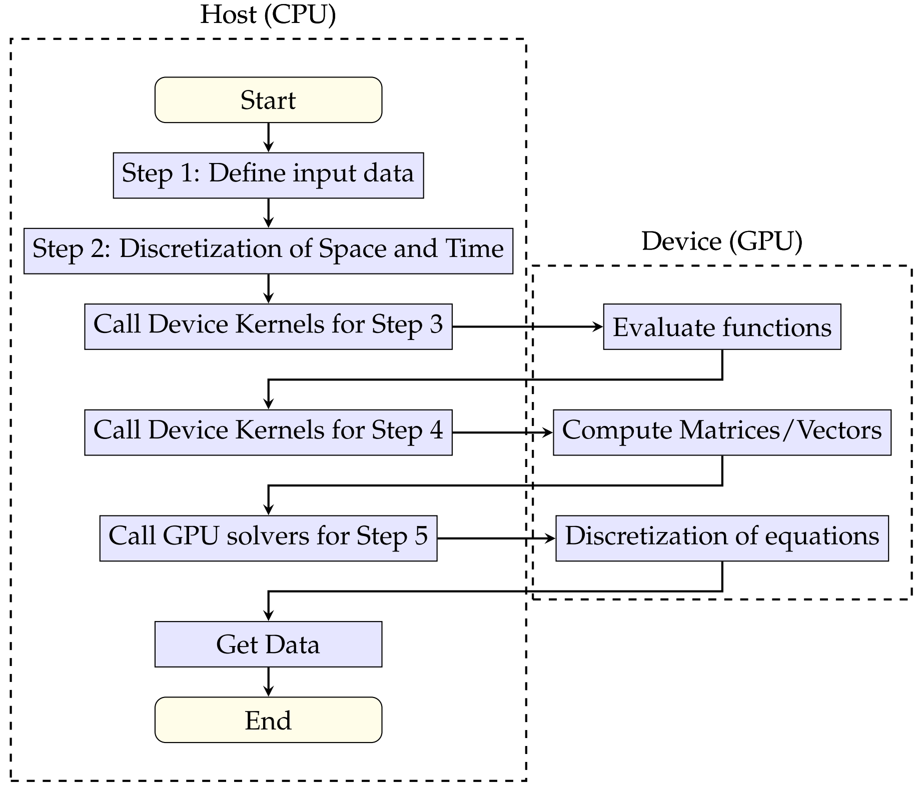


Figure 2. General flowchart for the numerical computation of the solution of (9)–(12) by applying the finite difference scheme (13) and (14).

Algorithm 1 Sequential Compute $\mathcal{F}$

1: **procedure** SEQUENTIALCOMPUTEEFE(x, t, L) ▷ Compute $\mathcal{F}$ sequentially
2: **Input:** Array x of spatial points, Array t of time points, Array L representing domain boundaries
3: **Output:** Matrix $\mathcal{F}$ containing the computed values
4: Initialize matrix $\mathcal{F}$ with zeros of size $(\text{length}(t), \text{length}(x))$
5: **for** i from 0 to $\text{length}(t) - 1$ **do**
6: **for** j from 0 to $\text{length}(x) - 1$ **do**
7: **if** $x[j] < L[1]$ **then**
8: $\mathcal{F}[i][j] \leftarrow f_1(x[j], t[i])$
9: **else if** $x[j] < L[2]$ **then**
10: $\mathcal{F}[i][j] \leftarrow f_2(x[j], t[i])$
11: **else**
12: $\mathcal{F}[i][j] \leftarrow f_3(x[j], t[i])$
13: **end if**
14: **end for**
15: **end for**
16: **end procedure**

Algorithm 2 Parallel Compute $\mathcal{F}$

```
1:  kernel compute_efe_kernel(x, t, L, F, t_len, x_len)            ▷ CUDA kernel
2:  n ← thread index in x dimension
3:  j ← thread index in t dimension
4:  if n < t_len and j < x_len then
5:      if x[j] < L[1] then
6:          F[n × x_len + j] ← f_1(x[j], t[i])
7:      else if x[j] < L[2] then
8:          F[n × x_len + j] ← f_2(x[j], t[i])
9:      else
10:         F[n × x_len + j] ← f_3(x[j], t[i])
11:     end if
12: end if
13: End kernel
14: procedure ThrowGPUKernel(x, t, L, F, t_len, x_len)
15: F ← allocate memory on GPU for matrix F
16: block_size ← (x_dimension, y_dimension, 1)          ▷ Dependency on hardware
17: grid_size ← calculate grid size based on t_len and x_len
18: call compute_efe_kernel(x, t, L, F, t_len, x_len, blocksize, gridsize)
19: return F                                    ▷ Pointer to the memory on GPU
20: End procedure
```

On the other hand, in Step 5, we note that the system's time evolution requires solving a linear system at each time step. This is achieved by precomputing the parameters for the system's right-hand side through calls to the GPU, as well as performing solver and norm computations to obtain residuals. A simple implementation that can be adapted for any built-in solver is presented in Algorithm 3. This general algorithm can be easily modified to use other solvers, whether implemented from scratch or available in libraries for GPU acceleration. In the next section, we analyze various solvers to evaluate their effectiveness in our context.

Algorithm 3 Evolution of the system

```
1:  function SIMPLEEVOLUTION(u_0, Â, B̂, A, B, C, s, N, len_x)
2:      Initialize matrix u with zeros of size (length(t), length(x))
3:      Initialize array residuals with zeros of size (length(t))
4:      u[0, :] ← d_u0
5:      rhs ← B̂u[0, :] + s[0, :]                         ▷ Calls GPU Operations
6:      u[1, :] ← solve(Â, rhs)                            ▷ Calls GPU Solver
7:      residuals[0] ← norm(Âu[1, :] − rhs)              ▷ Calls GPU Operations
8:      for n from 1 to N − 1 do
9:          rhs ← Bu[n, :] + Cu[n − 1, :] + s[n, :]       ▷ Calls GPU Operations
10:         u[n + 1, :] ← solve(A, rhs)                    ▷ Calls GPU Solver
11:         residuals[n] ← norm(Au[n + 1, :] − rhs)      ▷ Calls GPU Operations
12:     end for
13:     return u, residuals                      ▷ Pointers to the memory on GPU
14: end function
```

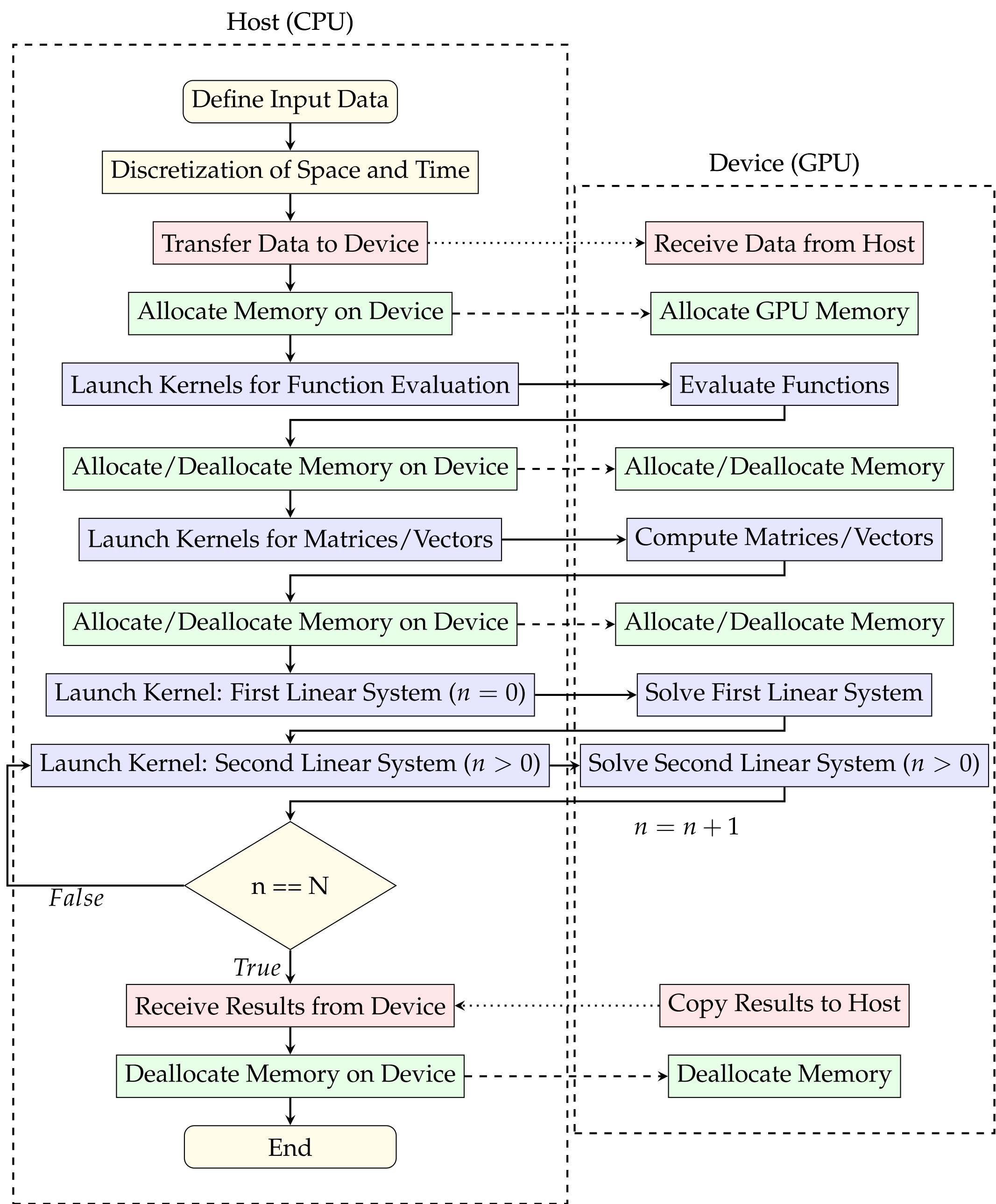


Figure 3. Specific flowchart for the numerical computation of the solution of (9)–(12) by applying the finite difference scheme (13) and (14).

4. Simulation Performance Using the GPU Accelerating Algorithm

4.1. Experimental Setup

The numerical simulations described in this study were conducted on a computer equipped with the following specifications:

- *Hardware Configuration*. Operating System: Ubuntu 22.04.03 LTS; Processor: 13th Gen Intel i5-13420H (12 cores); Memory: 16 GB DDR5 RAM; GPU: NVIDIA RTX4050 60 W; and Memory Size: 6 GB GDDR6.
- *Software Configuration*. The computations were accelerated using CUDA technology, and the following software tools and libraries were utilized: CUDA Compilation Tools: V12.3.107, PyCUDA Version: 2024.1, CuPy Version: 13.0.0, and Precision: float64.

This computational environment provided the necessary hardware and software resources to execute the numerical simulations. Some guidelines on the codes are in the Appendix A.

4.2. Physical and Mesh Parameters

In our simulation, we consider the physical parameters of Example 1 of [21]. The parameters are given in Table 5. The initial-boundary conditions and the source functions are defined as follows:

$$\psi_1(x) = \begin{cases} \sin(3\pi x/4), & x \in [0,1/3), \\ -\cos(3\pi(2x-1)/4) + \sqrt{2}, & x \in [1/3,2/3), \\ \cos(\pi(3x-1)/4), & x \in [2/3,1], \end{cases}$$
$$\psi_2(x) = -\frac{1}{2}\psi_1(x);$$
$$\varphi_1(t) = \varphi_2(t) = -3\pi \exp(-t/2)/8;$$
$$f(x,t) = \begin{cases} 8^{-1}(-2+9\pi^2)\exp(-t/2)\sin(3\pi x/4), & x \in [0,1/3), \\ 4^{-1}(1+9\pi^2)\exp(-t/2)\cos(3\pi(2x-1)/4), & x \in [1/3,2/3), \\ 8^{-1}(-2+9\pi^2)\exp(-t/2)\cos(\pi(3x-1)/4), & x \in [2/3,1]. \end{cases}$$

Then, the analytical solution of (9)–(12) is defined by:

$$u(x,t) = \begin{cases} \exp(-t/2)\sin(3\pi x/4), & x \in [0,1/3), \\ \exp(-t/2)\Big(-\cos(3\pi(2x-1)/4) + \sqrt{2}\Big), & x \in [1/3,2/3), \\ \exp(-t/2)\cos(\pi(3x-1)/4), & x \in [2/3,1]. \end{cases}$$

For the discretization of time and space, we consider several values of N and m_i:

$$N \in \{100, 200, 500, 1000, 2000, 5000, 10000\},$$
$$m_1 = m_2 = m_3 \in \{32, 64, 128, 256, 512, 1024, 2048\}.$$

Finally, the time range of the simulation is $T = 1$.

Table 5. Physical parameters for simulation.

	Layer 1 $\boldsymbol{\ell = 1}$	**Layer 2** $\boldsymbol{\ell = 2}$	**Layer 3** $\boldsymbol{\ell = 3}$
$\overline{W}_\ell$	1/3	1/3	1/3
C^ℓ	1	1	1
τ_q^ℓ	1	1	1
τ_T^ℓ	1	4	4/3
k^ℓ	4	1	6

4.3. Selection of Linear Solver Method

In this paragraph, we document the results of applying linear solvers to the discretized equations. Generally, linear solvers can be classified in direct and iterative methods. As a direct method, here, we use LU and QR, as iterative methods, we use the Jacobi and Conjugate Gradient method, experimenting also with preconditioning techniques. When using linear solvers for mathematical models, usually there are specific issues that need to be addressed: high condition numbers, particularly for fine spatial discretization, can severely impact convergence in iterative methods like Jacobi and CG, as they struggle with stability and require often robust preconditioning to stabilize. For direct methods, high condition numbers demand high precision (float64) to maintain stability, which increases memory demands, particularly on GPUs. Additionally, while LU and QR methods are robust and accurate, they are computationally intensive as matrix size grows, challenging

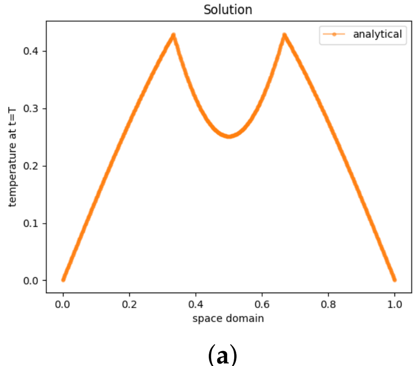
both memory and efficiency in large-scale models; though for sparse systems with special structures, the computational complexity can be kept low. Balancing precision, memory efficiency, and computational stability is essential for reliable, scalable solutions in this context. These issues are addressed in continuation when dealing with the matrix structure that arises from modelling heat conduction in a three-layered solid.

We recall that the matrices $\hat{A}$, $\hat{B}$, A, B, and C are tridiagonal, see Section 2. Additionally, we note that $\hat{B}$, B, and C are used to compute the system's right-hand side vector, so they do not impose any restrictions on the numerical method applied as a solver. However, the condition numbers of matrices $\hat{A}$ and A are excessively high for the required stability, as shown in Table 6. Although the condition number decreases as the size N increases, this rate of reduction is too slow to make larger values of N feasible for real, scalable applications. Moreover, the condition number increases with the number of spatial evaluation points, and higher values of m_i would necessitate even larger values of N to mitigate approximation errors.

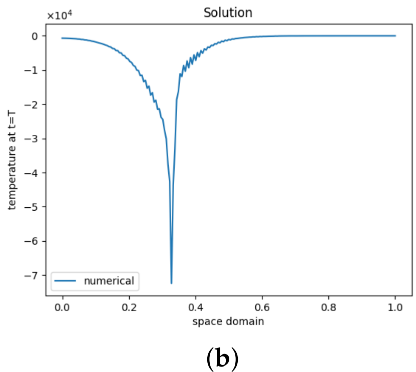
Table 6. Condition numbers of matrices A and $\hat{A}$. Here, $M_3 = m_1 + m_2 + m_3$ with $m_1 = m_2 = m_3$.

M_3 \ N	100	200	500	1000	2000	5000	10,000	100,000
96	A	4.8×10^4	2.7×10^4	1.1×10^4	6.3×10^3	3.3×10^3	1.4×10^3	7.5×10^2
	$\hat{A}$	3.7×10^4	2.3×10^4	1.2×10^4	6.5×10^3	3.4×10^3	2.2×10^3	1.1×10^3
192	A	3.8×10^5	2.1×10^5	9.7×10^4	5.1×10^4	2.7×10^4	1.1×10^4	5.7×10^3
	$\hat{A}$	2.9×10^5	1.8×10^5	9.1×10^4	5.0×10^4	2.6×10^4	1.1×10^4	5.7×10^3
384	A	3.1×10^6	1.7×10^6	7.7×10^5	4.1×10^5	2.1×10^5	8.9×10^4	4.5×10^4
	$\hat{A}$	2.3×10^6	1.4×10^6	7.2×10^5	4.0×10^5	2.16×10^5	8.8×10^4	4.5×10^4
768	A	2.4×10^7	1.3×10^7	6.2×10^6	3.3×10^6	1.7×10^6	7.1×10^5	3.6×10^5
	$\hat{A}$	1.9×10^7	1.2×10^7	5.8×10^6	3.2×10^6	1.6×10^6	7.1×10^5	3.5×10^5
1536	A	1.9×10^8	1.1×10^8	4.9×10^7	2.6×10^7	1.3×10^7	5.6×10^6	2.8×10^6
	$\hat{A}$	1.5×10^8	9.4×10^7	4.6×10^7	2.5×10^7	1.3×10^7	5.6×10^6	2.8×10^6
3072	A	1.5×10^9	8.7×10^8	3.9×10^8	2.1×10^8	1.1×10^8	4.5×10^7	2.3×10^7
	$\hat{A}$	1.2×10^9	7.5×10^8	3.7×10^8	2.0×10^8	1.0×10^8	4.5×10^7	2.3×10^7
6144	A	1.2×10^{10}	7.0×10^9	3.1×10^9	1.6×10^9	8.8×10^8	3.6×10^8	1.8×10^8
	$\hat{A}$	9.7×10^9	6.0×10^9	2.9×10^9	1.6×10^9	8.6×10^8	3.6×10^8	1.8×10^8

The effects of ill-conditioned matrices are well known. For instance, the evolution of the system using the Jacobi method diverges from the analytical solution. Figure 4 illustrates the known analytical solution (a) and the result of using the Jacobi method (b), where the error approaches 5×10^{93}. In none of the evaluated cases does the Jacobi method converge to the analytical solution.



(**a**)



(**b**)

Figure 4. Comparison between analytical solution and numerical solution using the Jacobi method: (**a**) Analytical solution, (**b**) Jacobi method with $m_i = 64$ and $N = 500$. The numerical temperature profile is clearly different from the analytic temperature profile. The inconsistency originated in the incorrect solution of the linear system by the selected linear solver.

The conjugate gradient (CG) method was also tested for solving the linear systems. Figure 5 shows the results of CG on two inputs, where the instability of the method is present. Specifically, an error of 95.04 was observed in case (a) and 5×10^{11} in case (b). Although CG provides a better approximation than Jacobi, the instability increases as the size of m_i grows. Additionally, several well-known preconditioners were tested with the iterative solvers. However, approaches as diagonal scaling, and incomplete Cholesky did not help the methods overcome the effects of high condition numbers.

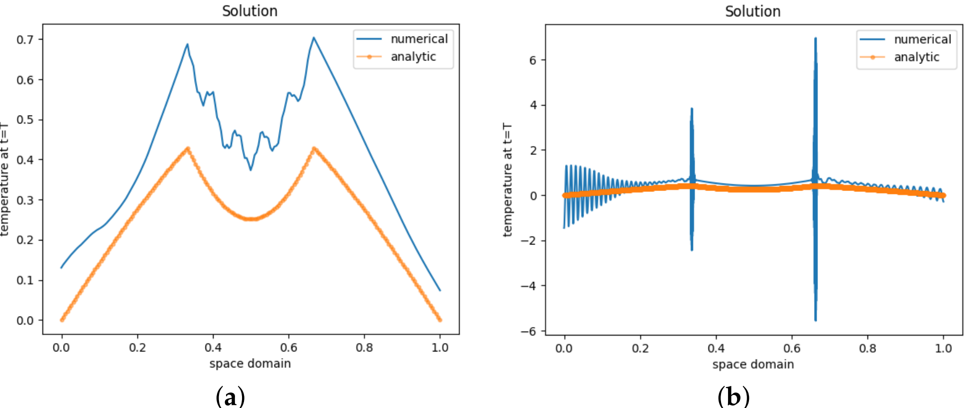


Figure 5. Numerical temperature profiles obtained with the conjugate gradient method, with (**a**) $m_i = 64$ and $N = 1000$, and (**b**) $m_i = 2048$ and $N = 5000$. The numerical temperature profiles are clearly different from the analytic temperature profiles. The figures show the inconsistency of the linear solver to approximate the linear system of the difference scheme.

While the iterative methods implemented diverge, the use of direct solvers, namely LU factorization and QR factorization, leads to the convergence of the system. Figure 6 shows the solution with LU and QR with $m_i = 2048$ and N = 10,000, where the error was approximately 4×10^{-3} and $1,3 \times 10^{-2}$, respectively. Note that the LU solution obtains a more precise approximation than QR, this is true for all the values of m_i and N.

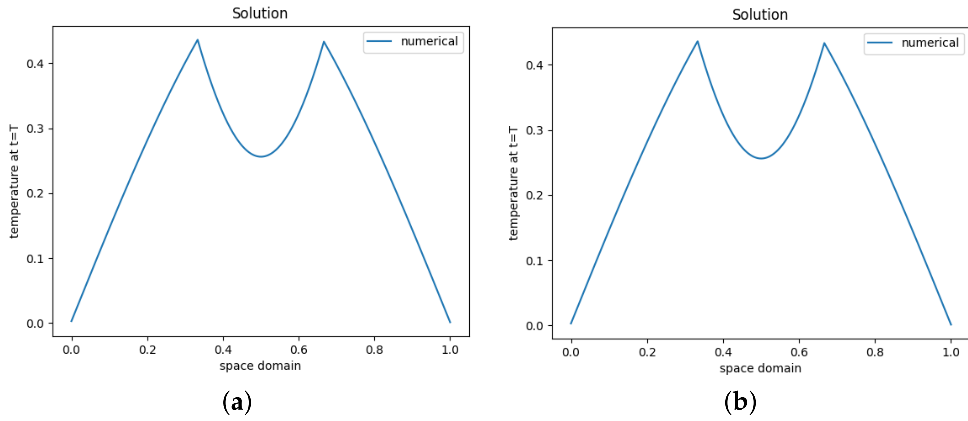


Figure 6. Temperature profiles for $m_i = 2048$ and N = 10,000 obtained with (**a**) LU method in the case, (**b**) QR method. The figure show that the numerical temperature profiles converges the analytic temperature profile when we consider the LU linear solver to approximate the linear system of the difference scheme.

To summarize, the LU and QR solvers converge, while the Jacobi, the conjugate gradient, the preconditioned Jacobi, and the preconditioned conjugate gradient methods diverge. In other words, the implemented direct methods converge, whereas the iterative methods fail to do so. In our analysis, the condition number appears to be the primary cause of divergence. However, further investigation into the choice of starting point and

the use of more specialized preconditioners could provide additional insights into this effect. For further examination, we consider only the methods that converge to the solution. The precision is set to float64; if reduced to float32, even the LU and QR methods diverge. We split the results into two sections: kernels implemented with PyCUDA (Steps 3 and 4) and the evolution of the linear system with CuPy (Step 5).

4.4. Accelerating with GPU: PyCUDA Acceleration for Steps 3 and 4

As mentioned in Section 4.1 on GPU kernels, these kernels are straightforward to construct given the characteristics of the functions. They do not entail data dependencies or synchronization that make the threads run slower; instead, matrix and vector values are computed independently. This allows the CUDA kernel to fully leverage the parallel capabilities of the GPU, as shown in Figure 7 presents. Note that the Speed-up is computed as $x = t_s/t_p$, where t_s is the sequential execution time and t_p the parallel execution time. For the largest instance ($m_i = 2048$ and N = 10,000), the CUDA kernel execution time is close to the order of milliseconds, significantly outperforming the CPU version, which requires up to hundreds of seconds. This reduction in computational time translates to a Speed-up of approximately 10^4 in Step 3 and 10^5 in Step 4. This acceleration is a natural outcome of the way the functions are constructed. Not only does this approach expedite the parameter computation, but it also keeps the data on the GPU, avoiding any data transfer between the CPU and GPU beyond the initial loading of discretization values in Step 2 and final data retrieval after Step 5. The average Speed-up for Steps 3 and 4, grouped by the value of m_i, is shown in Table 7.

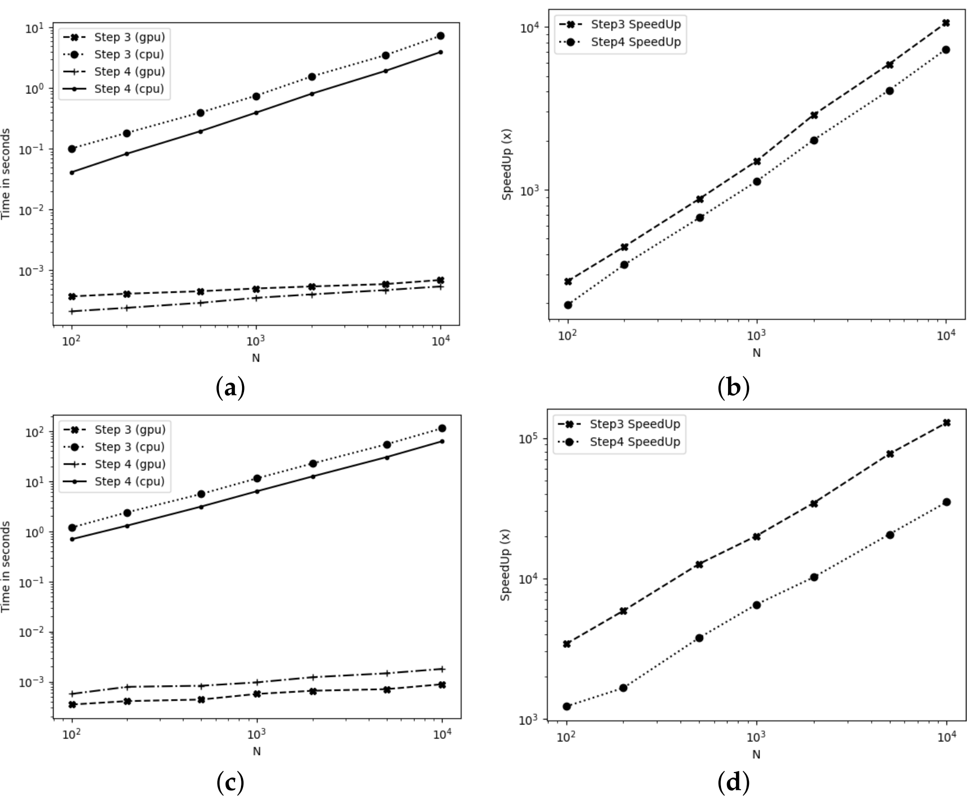


Figure 7. Comparison of computational times as N increases with a fixed value of m_i for steps 3 and 4 on GPU and CPU in logarithmic scale: (**a**) Time in seconds for $m_i = 128$, (**b**) Speed-up obtained with GPU for $m_i = 128$, (**c**) Time in seconds for $m_i = 2048$, (**d**) Speed-up obtained with GPU for $m_i = 2048$.

Table 7. Speed-up of both Steps 3 and 4 when using GPU with CUDA kernels. Here, $M_3 = m_1 + m_2 + m_3$ with $m_1 = m_2 = m_3$.

M_3	Speedup in Step 3	Speedup in Step 4
96	7.9×10^2	3.0×10^3
192	1.3×10^3	1.7×10^3
384	5.6×10^3	4.6×10^3
768	8.2×10^3	7.5×10^3
1536	1.6×10^4	1.3×10^4
3072	3.1×10^4	1.6×10^4
6144	5.5×10^4	1.8×10^4

4.5. Accelerating with GPU: Evolution of the System with CuPy

The *evolution* of the equation system is a more challenging step to parallelize on the GPU. The primary consideration is that, for each time step $n+1$ to be computed, the value from time step n is required. Thus, we are using a numerical method to solve the linear system with GPU assistance but are not fully leveraging parallelism by solving multiple linear systems simultaneously. This is the main limitation of this step, as discussed in Section 4.2.

The first optimization arises from the structure of both linear systems, as defined in (13) and (14), where the matrices $\hat{A}$ and A are constant for all time steps. This allows us to precompute the LU and QR factorizations once and then iterate through the time steps. Both CPU and GPU implementations use this approach of precomputing the factorization to enable a more accurate comparison of execution times.

Each linear system was solved using the precomputed factorization and the *solve_triangular* method from CuPy. We also tested the same approach with PyTorch, but found no notable difference in execution times, so the results presented here focus on the CuPy implementation. Figure 8 shows the execution times for two values of m_i, comparing LU and QR solvers on CPU and GPU hardware. The figure also presents the speed obtained for both solvers in both instances. For $m_i = 128$, QR on the CPU is significantly faster than the other cases for small values of N. The GPU implementations require more time than CPU on small instances, increasing their speed-up as the value of N grows. This is a common behaviour of highly parallel implementations, such as the ones done with GPU. For $m_i = 2048$, the GPU implementations are significantly better than the CPU solvers, reaching an approximate $5\times$ speed-up for LU and $6\times$ for QR. As the value of m_i increases the GPU exploits the parallelization over larger matrices and vectors, demonstrating improved acceleration.

Table 8 presents the average speed-up for all values of m. It shows that the system's evolution using the QR method can achieve a greater speed-up than with the LU solver. In both cases, the GPU becomes more efficient than the CPU on average when $m \geq 128$, with peak acceleration at $m = 2048$, the maximum value tested. Although the acceleration is higher for QR, LU is faster in most larger instances. The triangular solver with QR factorization benefits more from parallelism than the LU approach, but the LU-based solution performs better for larger cases. At the maximum values evaluated, $m_i = 2048$ and $N = 10^4$, QR is competitive with LU in terms of execution time. However, as noted in Section 4, LU achieves better accuracy than QR.

To conclude this section, we have several observations regarding the results shown in Figures 7 and 8. In Figure 7a,c, the chip type has a more significant impact than the step number. For both Steps 3 and 4, the GPU computation is significantly quicker, by a factor of more than 100, than CPU computation. For both Steps 3 and 4, GPU computation is over 100 times faster than CPU computation, with performance continuing to improve exponentially (linearly on an exponential scale) on larger grids. This improvement is due to the parallelization advantages that become more pronounced with larger grids.

Table 8. Speed-up comparison between LU and QR methods. Below the solid black line, the GPU implementation overtakes the CPU implementation. Here $M_3 = m_1 + m_2 + m_3$ with $m_1 = m_2 = m_3$.

M_3	LU Speedup	QR Speedup
96	0.489	0.547
192	0.723	0.817
384	1.101	1.314
768	1.708	2.234
1536	2.179	3.117
3072	2.846	3.873
6144	3.514	4.824

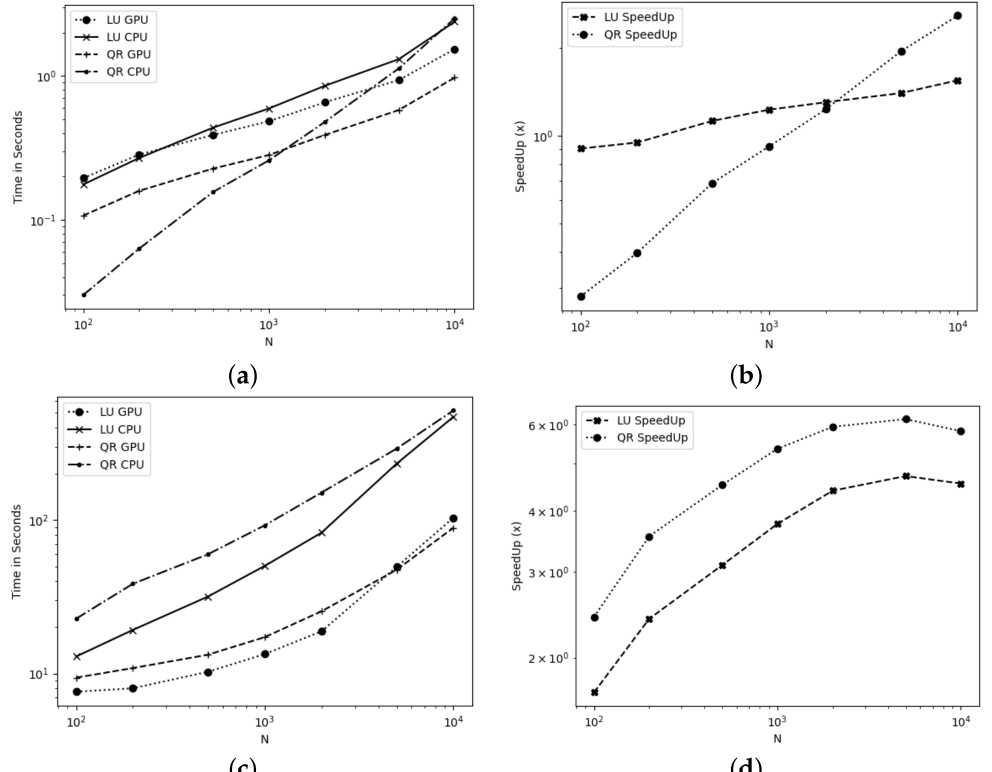


Figure 8. Comparison of computational times as N increases with a fixed value of m_i for LU and QR solvers on GPU and CPU in logarithmic scale: (**a**) Time in seconds for $m_i = 128$, (**b**) speed-up obtained with GPU for $m_i = 128$, (**c**) Time in seconds for $m_i = 2048$, (**d**) Speed-Up obtained with GPU for $m_i = 2048$.

In Figure 8a,b, for QR, if N is less than approximately 1100, the CPU performs better than the GPU; beyond this point, the GPU outperforms. GPU use requires some overhead, such as data transmission, independent of grid size, which impacts performance on smaller grids. Figure 8c shows that the QR method, by design, requires additional computational time to calculate the Q matrix. Figure 8d illustrates that QR speed-up is greater than LU speed-up, as QR is a more computationally intensive algorithm and, thus, has more potential for improvement.

5. Discussion on Findings, Implications and Future Work

In this discussion section, we first summarize the findings, then outline the implications, and finally give perspectives for future work.

5.1. Findings

5.1.1. Efficient GPU Usage

We have analyzed different numerical methods for solving the evolution of the linear system in the context of a three-phase-lag solid and found that LU and QR solvers are suitable for approximating the solution correctly. We also demonstrated that the matrices and vectors required for the linear system can be computed efficiently with customized CUDA kernels to meet the problem specifications. Our approach exploits GPU parallel capabilities, reducing the time from seconds to milliseconds, with the maximum acceleration obtained in the order of 10^4.

5.1.2. Acceleration Factors

We studied the model and identified the key points of parallelization and the challenges. We found that the parallelization cannot be conducted across the temporal domain, instead we focused on computing each time step sequentially and accelerating the solution for each linear system. Considering that, the maximum speed-up obtained was approximately $5\times$ for LU and $6\times$ for QR. The acceleration obtained with the use of GPU increased proportionally with the size of the problem, both in terms of spatial and temporal discretization.

5.1.3. Accuracy and Stability

Our experiments have shown that the LU solver consistently delivers higher accuracy than the QR solver, while both remain competitive in terms of efficiency. It is important to note that the speed-up obtained by QR, while larger than that of LU, is relative to the overall time required by the methods. In fact, in most cases, LU requires less time than QR, reinforcing the competitive efficiency of both solvers.

For the accuracy of the solutions, we saw that the iterative methods implemented resulted in divergence or instability. The use of some well known preconditioners did not make the methods converges; therefore, a more in-depth study on specialized preconditioners is proposed as a future work to continue with research.

5.2. Implications

What are the implications of our findings for GPU-accelerated algorithms in heat conduction simulations? Briefly, there are both the strengths and limitations of GPU architectures when applied to large-scale thermal models governed by differential equations and discretized into high-dimensional, sparse linear systems.

5.2.1. Real-Time Simulations in Thermal Modeling

A key finding is the ability of GPU-accelerated algorithms to perform high-resolution simulations at a fraction of the time required by traditional CPU-based methods. By reducing computation times from seconds to milliseconds, GPUs allow researchers and engineers to conduct real-time or near-real-time heat conduction analyses. This capability opens new possibilities for applications where time-sensitive thermal modeling is crucial, such as in transient heat flux monitoring for electronic devices, rapid prototyping in materials design, or adaptive temperature control in manufacturing processes. This becomes even more relevant in optimization setups, where best parameters need to be found by simulating in short time a series of scenarios with varying parameters.

GPU-enabled scalability spatial and temporal granularity without significantly increasing computational overhead is particularly valuable in fields like materials science, where understanding heat flow across complex, layered materials requires fine-grained analysis of heat gradients, interfaces, and boundary interactions. Overcoming otherwise limited computational resources simulations to be conducted with high fidelity, offering deeper insights into thermal behavior in real-time applications or large-scale simulations. Specifically, GPU-accelerated heat transfer model for multi-layered solids can directly inform simulations in various applications contexts, e.g., fabric-type actuators [30], where

distinct thermal properties across flexible material layers influence heat dissipation critical for safe, efficient operation.

5.2.2. Solver Selection for Systems with High Condition Numbers

In particular, we addressed the issue of refining solver strategies specifically for heat conduction problems with high condition numbers. The study reveals that direct solvers (LU and QR) are not only more stable, but also perform better than iterative methods (Jacobi and conjugate gradient) in GPU-accelerated environments for high-condition-number matrices typical of fine spatial discretization. This insight is relevant for multi-layered materials where heat conductivity varies across interfaces, which commonly produces ill-conditioned systems. By identifying the limits of iterative methods in handling large condition numbers, this research points to direct solvers as the more reliable and accurate choice. Thus further investigation into specialized iterative preconditioning approaches or hybrid methods that could unlock iterative solvers' potential can be encouraged.

5.2.3. Precision Considerations

A key finding is the the need for double precision (float64) as an essential consideration for scientific computing on GPUs, in particular to compensate the effects of high condition numbers: while GPUs offer substantial speed-ups, they require careful calibration of precision settings to maintain accuracy for scientific models that are sensitive to numerical stability. The need for precision management in GPU computing raises the question of developing adaptable mixed-precision schemes: Mixed-precision algorithms could, for example, use double precision selectively for parts of the computation most susceptible to instability, while using single precision elsewhere to enhance speed.

5.2.4. Framework for GPU-Optimized Finite Difference Methods

From the successful use of GPU-accelerated approaches to finite difference methods, this study establishes a framework for extending GPU utilization to broader types of parabolic PDEs, especially those that exhibit similar structural characteristics in terms of sparsity and boundary conditions. By analyzing the limitations of iterative methods, this study paves the way for a new class of GPU-optimized solvers tailored to specific computational challenges for solving PDEs. This points towards the potential for algorithmic innovation—such as hybrid approaches combining direct and iterative techniques or iterative solvers adapted for multi-GPU environments that might alleviate the condition number issues encountered.

5.3. Future Work

As a conclusion, we can identify two promising avenues for future research and optimization. The first involves the design and implementation of specialized solvers using PyCUDA, which can exploit the unique characteristics of the linear systems and the parallelization capabilities of GPUs. The second approach is to delve into the mathematical aspects of the model and design a numerical scheme that can better leverage the parallelization on GPUs. These potential optimizations offer hope for further advancements in the field. Consequently, in our future work, we aim to extend the results of the present research in at least three ways.

5.3.1. High Condition Numbers

First, we have observed that the matrices involved in the finite difference scheme have large condition numbers as the number of discretization nodes in the spatial domain increases (see Table 6). To address this, we plan to investigate advanced linear solution methods, such as hybrid methods, which have shown high precision and stability in handling transient heat conduction problems [31], as well as network-based methods with observer systems [32].

5.3.2. Multidimensional Geometries

Second, we will explore extensions of the heat transfer model to multidimensional problems or more complex geometries [33,34]. This step will build on the current one-dimensional model and expand our approach to include simulations in higher dimensions.

5.3.3. Validation by Parameter Identification

Third, for the model validation for specific applications, we intend to develop a numerical parameter identification process using data from published experimental results. This approach will allow us to refine model parameters, aligning simulations more closely with experimental observations.

5.3.4. Outlook on Scalability

While this study focused on a proof of concept for employing GPU-acceleration in a finite-difference setup, the methods used are designed to be compatible with multi-GPU setups or HPC clusters. Future work might address scaling optimizations such as memory partitioning, parallel kernel execution, and inter-GPU communication efficiency to enhance the model's applicability in large-scale simulations.

Author Contributions: Conceptualization, A.C. and S.B.; methodology, A.C.; software, N.M.; validation, N.M.; formal analysis, N.M., A.C. and S.B.; investigation, F.H.; resources, A.C. and A.T.; data curation, F.H.; writing—original draft preparation, A.C. and A.T.; writing—review and editing, A.C., A.T. and S.B.; visualization, F.H.; supervision, A.C. and S.B.; project administration, A.C. and S.B.; funding acquisition, A.C. and A.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Universidad del Bío-Bío (Chile); National Agency for Research and Development, ANID-Chile, through FONDECYT project 1230560 and FONDEF project ID23I10026; VRIIP of Universidad de Antofagasta, and by the Competition for Research Regular Projects, year 2023, code LPR23-03, Universidad Tecnológica Metropolitana.

Data Availability Statement: The original files used to produce the simulations in the present work are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Short Supplement on GPU Computing

In GPU computing, parallel acceleration is achieved by dividing computations into smaller tasks that are processed simultaneously across thousands of GPU cores, which are highly efficient at handling parallel tasks. This approach has been outlined in previous sections for our heat conduction model, where matrix and vector operations for linear solvers in discretized models are effectively parallelized. The implementation is detailed down to the source code level, available in the repository: https://github.com/nrmurua/heat1d (accessed on 2 November 2024).

Custom kernel programming provides fine-grained control over parallel operations on the GPU, allowing us to achieve substantial performance gains. Here, we illustrate this process using the calculation of $\mathcal{F}$, an approximation of $f(x,t)$.

Appendix A.1. Computing—Solver Level

At the top level (in the file *gpu_heat.py*), the main computation for $\mathcal{F}$ is performed with:

```
d_Efe = gpu.gpu_compute_efe(d_x, d_t, d_L, len_t, len_x)
```

This variable is then used within the model logic, in this case for calculating the right-hand side vector:

```
d_Nsource = gpu.gpu_compute_Nsource(Dt, d_Dx, ..., d_Efe, ..., N, len_x)
```

After computation, memory is deallocated to free up GPU resources:

```
d_Efe.free()
```

The right-hand side vector is subsequently passed into a specific solver:

```
cupy_lu_solver(d_u0, d_Nsource, d_hA, d_hB, d_A, d_B, d_C, N, len_x)
```

This solver is called when comparing the performance of different solvers:

```
u, residuals = switch_solver(s, d_u0, d_Nsource, ..., N, len_x)
```

Appendix A.2. Computing—Matrix Level

For computing individual vector components, a subroutine is implemented in *gpu_utilities.py*:

```
def gpu_compute_efe(d_x, d_t, d_L, len_t, len_x):
```

This function handles memory allocation, block and grid size specification, and delegates the actual computation to a CUDA kernel in *kernels_utilities.py*. Here, the effective assignment of values to the vector occurs within the CUDA kernel function `compute_efe_kernel`, which is defined in Python as `cuda_compute_efe`.

Similar calculations are carried out for all other matrices and vectors involved in the model, ensuring efficient use of GPU resources and optimal parallel performance.

References

1. Dutil, Y.; Rousse, D.R.; Salah, N.B.; Lassue, S.; Zalewski, L. A review on phase-change materials: Mathematical modeling and simulations. *Renew. Sustain. Energy Rev.* **2011**, *15*, 112–130. [CrossRef]
2. Assis, E.; Ziskind, G.; Letan, R. Numerical and experimental study of solidification in a spherical shell. *J. Heat Tran.* **2009**, *131*, 024502. [CrossRef]
3. Aydın, O. Conjugate Heat Transfer Analysis of Double Pane Windows. *Build. Environ.* **2006**, *41*, 109–116. [CrossRef]
4. Ehms, J.H.N.; Oliveski, R.D.C.; Rocha, L.A.O.; Biserni, C.; Garai, M. Fixed grid numerical models for solidification and melting of phase change materials (PCMs). *Appl. Sci.* **2019**, *9*, 4334. [CrossRef]
5. Liu, X.Y.; Xie, Z.; Yang, J.; Meng, H.J. Accelerating phase-change heat conduction simulations on GPUs. *Case Stud. Therm. Eng.* **2022**, *39*, 102410. [CrossRef]
6. Raimundo, A.M.; Oliveira, A.V.M. Assessing the impact of climate changes, building characteristics, and HVAC control on Energy Requirements under a Mediterranean Climate. *Energies* **2024**, *17*, 2362. [CrossRef]
7. Jezierski, W.; Święcicki, A.; Werner-Juszczuk, A.J. Deterministic mathematical model of energy demand of single-family building with different parameters and orientation of windows in climatic conditions of Poland. *Energies* **2024**, *17*, 2360. [CrossRef]
8. Korkut, T.B.; Rachid, A. Numerical investigation of interventions to mitigate heat stress: A case study in Dubai. *Energies* **2024**, *17*, 2242. [CrossRef]
9. Tan, L.; Gao, D.; Liu, X. Can Environmental information disclosure improve energy efficiency in manufacturing? Evidence from Chinese Enterprises. *Energies* **2024**, *17*, 2342. [CrossRef]
10. Walacik, M.; Chmielewska, A. Energy performance in residential buildings as a property market efficiency driver. *Energies* **2024**, *17*, 2310. [CrossRef]
11. Koshlak, H.; Basok, B.; Davydenko, B. Heat transfer through double-chamber glass unit with low-emission coating. *Energies* **2024**, *17*, 1100. [CrossRef]
12. Jezierski, W.; Zukowski, M. Evaluation of the impact of window parameters on energy demand and CO_2 emission reduction for a single-family house. *Energies* **2023**, *16*, 4429. [CrossRef]
13. Tzou, D.Y. *Macro to Microscale Heat Transfer. The Lagging Behaviour*, 2nd ed.; Taylor & Francis: Washington, DC, USA, 2014.
14. Dai, W.; Han, F.; Sun, Z. Accurate numerical method for solving dual-phase-lagging equation with temperature jump boundary condition in nano heat conduction. *Int. J. Heat Mass Transf.* **2013**, *64*, 966–975. [CrossRef]
15. Jain, A.; Krishnan, G. Stability analysis of a multilayer diffusion-reaction heat transfer problem with a very large number of layers. *Int. J. Heat Mass Transf.* **2024**, *231*, 125769. [CrossRef]
16. Jain, A.; Krishnan, G. Thermal stability of a two-dimensional multilayer diffusion-reaction problem. *Int. J. Heat Mass Transf.* **2024**, *221*, 125038. [CrossRef]
17. Bandhauer, T.M.; Garimella, S.; Fuller, T.F. A critical review of thermal issues in lithium-ion batteries. *J. Electrochem. Soc.* **2011**, *158*, R1. [CrossRef]
18. Hickson, R.I.; Barry, S.I.; Mercer, G.N.; Sidhu, H.S. Finite difference schemes for multilayer diffusion. *Math. Comput. Model.* **2011**, *54*, 210–220. [CrossRef]
19. March, N.G.; Carr, E.J. Finite volume schemes for multilayer diffusion. *J. Comput. Appl. Math.* **2019**, *345*, 206–223. [CrossRef]

20. Zhou, Y.; Wu, X.Y. Finite element analysis of diffusional drug release from complex matrix systems. I.: Complex geometries and composite structures. *J. Control. Release* **1997**, *49*, 277–288. [CrossRef]
21. Coronel, A.; Lozada, E.; Berres, S.; Fernando, F.; Murúa, N. Mathematical modeling and numerical approximation of heat conduction in three-phase-lag solid. *Energies* **2024**, *17*, 2497. [CrossRef]
22. Demmel, J.W. *Applied Numerical Linear Algebra*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1997.
23. Trefethen, L.N.; Bau, D. *Numerical Linear Algebra*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2022.
24. LeVeque, R.J. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2007.
25. Van der Vorst, H.; Van Dooren, P. (Eds.) *Parallel Algorithms for Numerical Linear Algebra*; Elsevier: Amsterdam, The Netherlands, 2014.
26. Okuta, R.; Unno, Y.; Nishino, D.; Hido, S.; Loomis, C. Cupy: A numpy-compatible library for nvidia gpu calculations. In Proceedings of the Workshop on Machine Learning Systems (LearningSys), in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS), Long Beach, CA, USA, 4–9 December 2017; Volume 6.
27. Klöckner, A.; Pinto, N.; Lee, Y.; Catanzaro, B.; Ivanov, P.; Fasih, A. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* **2012**, *38*, 157–174. [CrossRef]
28. Sun, H.; Sun, Z.Z.; Dai, W. A second-order finite difference scheme for solving the dual-phase-lagging equation in a double-layered nano-scale thin film. *Numer. Methods Partial. Differ. Equ.* **2017**, *33*, 142–173. [CrossRef]
29. Coronel, A.; Huancas, F.; Lozada, E.; Tello, A. A numerical method for a heat conduction model in a double-pane window. *Axioms* **2022**, *11*, 422. [CrossRef]
30. Peng, Y.; Yamaguchi, H.; Funabora, Y.; Doki, S. Modeling Fabric-Type Actuator Using Point Clouds by Deep Learning. *IEEE Access* **2022**, *10*, 94363–94375. [CrossRef]
31. Sun, W.; Ma, H.; Qu, W. A hybrid numerical method for non-linear transient heat conduction problems with temperature-dependent thermal conductivity. *Appl. Math. Lett.* **2024**, *148*, 108868. [CrossRef]
32. Guarro, M.; Ferrante, F.; Sanfelice, R.G. A hybrid observer for linear systems under delayed sporadic measurements. *Int. J. Robust Nonlinear Control.* **2024**, *34*, 6610–6635. [CrossRef]
33. Cao, R.; Chen, Y.; Shen, N.; Li, H.; Chen, S.; Mai, Y.; Guo, F. Enhancing Spectral Response of Thermally Stable Printed Dion–Jacobson 2D FAPbI3 Photovoltaics via Manipulating Charge Transfer. *ACS Energy Lett.* **2024**, *9*, 3737–3745. [CrossRef]
34. Mubaraki, A.M.; Nuruddeen, R.I. Steady-state thermodynamic process in multilayered heterogeneous cylinder. *Open Phys.* **2024**, *22*, 20240067. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.