Efficient parallel implementation of a transient heat transfer model

Tim Christian Axel Westerlund

School of Chemical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo January 25, 2018

Supervisor

Prof. Ari Jokilaakso

Advisors

Prof. Jukka Suomela

D.Sc. Seppo Louhenkilpi
# Abstract

A three-dimensional slab furnace model was optimized using various parallelization techniques, memory access pattern improvements and preprocessing optimizations. This work both optimized the existing version of the model and implemented a new version of the model that utilized the graphics processing unit (GPU). The GPU version was observed to be $6 \times$ faster than the baseline, optimized CPU (.NET/C#) version $2 \times$ faster and CPU (CLI/C++) version $3 \times$ faster than the baseline. The scaling with respect to amount of nodes was noted to be nearly linear, with the GPU implementation scaling the best out of all the versions of the model.

**Keywords** GPU, FDM, heat transfer model, parallel algorithms, process control
Tekijä Tim Christian Axel Westerlund

Työn nimi Efficient parallel implementation of a transient heat transfer model

Koulutusohjelma Materiaaliteknikka

Pääaine Toiminnalliset materiaalit

Pääaineen koodi CHEM3025

Työn valvoja Prof. Ari Jokilaakso

Työn ohjaaja Prof. Jukka Suomela, TkT Seppo Louhenkilpi

Päivämäärä 25. tammikuuta 2018

Sivumäärä 48+3

Kieli Englanti

Tiivistelmä


Avainsanat GPU, rinnakkaislaskenta, lämmönsiirto, FDM, prosessiohjaus, mallinnus
Preface

This thesis mirrors my studies throughout my bachelor’s and master’s degree studies alike: working on this thesis allowed me to bridge my knowledge and interest between materials science and computer science. Out of the computer science courses I completed, two courses stood out: Programming Parallel computers and Computer Graphics. In this thesis, I was able to use some of my knowledge acquired during the former of those two and combine it with my experience in materials science. Combining these fields is something I had wanted to do during my studies, but was not able to do so prior to working on this thesis.

I want to thank my supervisor, Prof. Ari Jokilaakso, for the opportunity to complete this work within his research group. I also want to thank him for continued feedback on the thesis and for helping on the practical issues I ran into throughout the period I was working on this thesis.

I want to thank my advisor Prof. Jukka Suomela for his continued feedback throughout this thesis work, our fruitful discussions which resulted in many ideas on how to improve the model as well as figuring out potential issues within the model. I want to also thank him for introducing me to parallel programming earlier in 2017, when I attended his course on programming parallel computers.

I want to thank my advisor D.Sc. Seppo Louhenkilpi for his continued feedback and for his help on understanding the importance of such models from the metallurgical perspective.

I want to thank M.Sc. Jukka Laine for providing this opportunity to work with Casim Consulting Oy as well as for his participation in discussions regarding the model. I would also like to extend my gratitude to M.Sc. Risto Vesanen, without whom this thesis would not have been written. I want to thank him for being willing to share the model he had prepared with Casim Consulting and for our discussions about improving the model. My knowledge and understanding of the mathematic concepts presented in this thesis are far deeper than they would be without him.

This thesis was funded by TEKES and has been completed as part of a larger project: FLEX Flexible and Adaptive Operations in Metal Production (8155/31/2016).

Otaniemi, January 25, 2018

Tim Christian Axel Westerlund
## Contents

Abstract  ii  
Abstract (in Finnish)  iii  
Preface  iv  
Contents  v  
Symbols and abbreviations  vii  

1 Introduction  1  

2 Background  3  
2.1 The reheating process  3  
2.2 The heat transfer equation and discretization  5  
2.2.1 The enthalpy method  7  
2.2.2 Explicit methods  7  
2.2.3 Implicit methods  9  
2.2.4 Crank–Nicolson method  10  
2.3 Iterative methods: Jacobi and modified Gauss–Seidel method  11  
2.4 Parallel computing  12  
2.4.1 Parallelization in modeling  14  

3 Research material and methods  16  
3.1 The slab furnace model  16  
3.2 The CUDA programming model  20  
3.3 Validation  22  
3.4 Evaluation of the present implementation of the model  23  
3.5 Analysis on parallelization possibilities of the existing model  25  
3.5.1 Time complexity of current implementation  26  
3.5.2 Theoretical running times of parallelized implementation  26  
3.5.3 Host code (CPU) optimization  28  
3.5.4 GPU-kernel implementation  28  
3.6 Benchmarking  31  

4 Results  32  
4.1 Parallel efficiency  38  

5 Discussion  40  

6 Summary and conclusions  43  
6.1 Future work  43  

References  45  

Symbols and abbreviations

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEM</td>
<td>Boundary-element method</td>
</tr>
<tr>
<td>BTCS</td>
<td>Backward-Time Central-Space</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>FDM</td>
<td>Finite Difference Method</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite Element Method</td>
</tr>
<tr>
<td>FLOP(s)</td>
<td>Floating Point Operation (per second)</td>
</tr>
<tr>
<td>FTCS</td>
<td>Forward-Time Central-Space</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General-purpose computing on graphics processing units</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
</tbody>
</table>

Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_p$</td>
<td>Specific heat capacity</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Emissivity</td>
</tr>
<tr>
<td>$k$</td>
<td>Thermal conductivity</td>
</tr>
<tr>
<td>$h$</td>
<td>Heat transfer coefficient</td>
</tr>
<tr>
<td>$T$</td>
<td>Temperature</td>
</tr>
<tr>
<td>$H(T)$</td>
<td>Enthalpy function</td>
</tr>
<tr>
<td>$K(T)$</td>
<td>Kirchhoff function</td>
</tr>
<tr>
<td>$Q$</td>
<td>boundary mobility activation energy</td>
</tr>
</tbody>
</table>
1 Introduction

The aim of this work is to optimize and implement a transient heat transfer model in order to speed up the calculation process for industrial purposes. The intended purpose of the model is to control a walking beam reheating furnace for steel slabs. The model is to be utilized in at least two ways: to verify that the temperature distribution is uniform enough for the process to go to next step and to feed temperature information to other process controlling models that rely on the temperature information. Currently, the heating times in the furnace are quite long to make sure that the temperature distribution is uniform and that the entire slab is at the desired temperature; the heating time could potentially be reduced with greater knowledge of the temperature distribution. The process of heating slabs is usually the next step after the slabs have been produced via continuous casting.

Heating the slabs typically takes place in walking beam furnace, where the slabs are heated through both convective and radiative heat transfers [1], which means that both heat transfer methods must be considered in the model. The multiple heat transfer mechanisms are taken into consideration as boundary conditions that the heat transfer model has to consider. The resulting heat transfer equations can then be solved either analytically or numerically. Usually the governing equations of such phenomena tend to be differential equations. For simple systems, analytical solutions can be found; for more complex systems, discretization of differential equations is usually required. Most commonly used numerical methods are finite difference method (FDM), finite volume method, finite element method (FEM), boundary-element method (BEM) and various meshless and gridless methods [2].

This work takes an existing three dimensional finite difference heat transfer model and aims to optimize it. The baseline version has been built by Casim Consulting Oy. The baseline version of the model includes test data for a specific process and material data for the steel slabs used in that process, as well as a CPU implementation of the model that is partially parallelized. The model is used to calculate temperatures within a steel slab during reheating process and the purpose of the model is to aid in process control. The portions that are parallel in the baseline version are the iterative functions that are ran on every time step. The baseline version contains a test project that can be used to initialize and commence computation, a solver class that contains the critical parts where heat transfer is calculated and a slab class that reads and contains the material specific parameters for the slab. As the model is intended to be used as an online industrial model, the model will receive temperature data from within the furnace on each time step that are required for the computation of the boundary conditions.

This work optimizes the existing CPU implementation heat transfer model and additionally implements the same model on graphics processing unit (GPU). The tool chosen for this project is CUDA, which is a platform for general purpose computing on graphics processing units (GPGPU) programming. Optimization in this
context is defined as reduction in computation time. The model is optimized via parallelization and other available methods such as improved memory access patterns, manipulation of data structures and improved algorithms where possible. When the implicit form of FDM is utilized, solving a large number of linear equations is required. This model utilizes an implicit (Crank–Nicolson) discretization scheme, where information is utilized both from current and previous time step. Explicit scheme, which only utilizes information from the previous time step, can be faster per single time step, but it has been shown that the explicit schemes are often outpaced by implicit methods due to their increased stability, which allows implicit methods to utilize much larger time steps than explicit methods [2]. Modern graphical processing units offer massively parallel computing possibilities, which are well suited to solving problems such as this one [3].
2 Background

Many metallurgical processes require knowing the exact temperature of any given object, at any point, at any time. One way to accomplish that is to calculate the temperature gradients using various adaptations of the heat transfer equation (See section 2.1). For simple cases analytical solutions can be found, however in any given production environment there are various external factors that favor numerical methods as varying boundary conditions are easier to implement. These external factors can include varying boundary conditions, temperature dependent material properties and heterogeneous materials. Numerical solutions however can be found for almost any case given the material parameters and environment information. In the past the viability of numerical methods has been limited by computational capability. The most computationally taxing part of solving implicit FDM models is the solving of large systems of linear equations. The computationally expensive portion of calculating explicit schemes is either from small time steps or largely increased mesh density. The immense parallel computing power that modern GPUs have to offer is a very promising solution to this problem [4].

2.1 The reheating process

The reheating process of slabs usually takes place after continuous casting of the slabs and before the hot rolling process. There is usually quality inspection in between each of the phases. The temperature of the slab while entering the furnace is dependent on how long it has been let to cool after casting and the temperature of the slab exiting the furnace is dependent on the needs of the hot rolling process [1, 5]. The process flowchart can be seen in Figure 1.

![Flowchart of the process.](image)

The furnace used in this process is a walking beam furnace where slabs are placed onto stationary rails and are then rotated by the moving beams, which move the slab from one stationary point to another on constant intervals. The furnace also contains temperature sensors, which can be utilized to produce a distribution of the temperature of the air within the furnace as well as the temperature of the walls. These values can then be interpolated to produce the external values needed by the model used in this work. For the purpose of this work, a simple schematic of a walking beam furnace can be seen in Figure 2.
The main purpose of heating a steel slab to high temperatures, for example to 1200 °C is to allow for ease of plastic deformation of the steel; to make it more malleable for hot rolling [6, 7]. Reheating the slabs consumes a large amount of energy; it has been estimated that reheating furnaces are responsible for 67 % of total energy consumption of an integrated steel plant [6]. Therefore the energy efficiency of such furnace is quite important and thus it has seen a variety of research attempting to optimize the efficiency. Many processes involve a quality check with various material characterization methods after the casting has been completed. As most of the materials characterization methods are only suitable in low temperatures, the slabs are often cooled down to room temperature or alternatively the slabs can be cooled down in controlled environment to an uniform temperature. There is a possibility that accurate models could predict quality of steel during the continuous casting, which would allow the slabs to be placed to reheating furnace at higher temperatures or even possibly eliminate the need for the reheating furnace altogether. Some processes already involve direct hot rolling after continuous casting [8].

As the next step is hot rolling, where the slab is rolled with pairs of rolls to reduce thickness, it is important that the slab deforms uniformly. As the malleability is highly dependent on the temperature [7, 9], currently a lot of processes without precise process control software keep the slabs in the reheating furnace for extended periods of time just to make sure that the temperature distribution is uniform. However, such long periods in extended periods can start to cause unwanted grain growth, which can affect the mechanical properties of the material in a negative way [7]. The rate of ideal grain growth [7] is determined as

$$d^2 - d_0^2 = kt,$$  \hspace{0.5cm} (1)

where $d_0$ is the initial grain size, $d$ is the final grain size and $k$ is temperature
dependent constant defined as $k = k_0 \exp(-Q/RT)$, where $T$ is the temperature, $k_0$ is experimentally obtained material dependent pre-exponential constant, $R$ is the gas constant and $Q$ is the boundary mobility activation energy. That is, to reduce grain growth we can either decrease temperature which is not possible in this case or decrease the time spent at elevated temperature. In the future, reheating time could be reduced which could result in less grain growth during the reheating process and more energy efficient process overall. The less time there is between continuous casting and reheating for hot rolling, the less time is taken to heat the material which could yield improved mechanical properties.

Ideally, the slabs would be placed directly to reheating furnace after continuous casting to equalize the temperature distribution for hot rolling. This would reduce the heating time, and therefore might slightly reduce the grain growth that inevitably occurs during heating. Usually the temperature of the steel is slightly too low after the continuous casting, for it to be directly hot rolled, but a theoretical possibility would be to add some heating elements towards the end of the continuous casting process, which could allow a more streamlined process.

### 2.2 The heat transfer equation and discretization

Most FDM heat transfer discretizations are derived from the same parabolic differential heat conduction equation. The simplest three dimensional form of the equation is

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

where $u(x,y,z,t)$ is the value that the equation solves and $\alpha$ is the diffusivity coefficient that is defined as $k/\rho c_p$, where $k$ is thermal conductivity, $\rho$ is density and $c_p$ is specific heat capacity [2].

The equation can be reduced to two or one dimension with removing the $y$ and $z$-terms. For simplicity, the following discretization steps will be done with two-dimensional systems. The first step of the discretization process is to map $u(x,y)$ to $u(i\Delta x, j\Delta y)$, where $\Delta x$ and $\Delta y$ replace the continuous $(x,y)$ coordinates, while $i$ and $j$ represent the finite steps in space. For example $u_{i+1,j} = u(x_0 + \Delta x, y_0)$ [2].

The spatial finite difference approximation can be derived from the definition of derivative for two dimensional function:

$$\frac{\partial u}{\partial x} = \lim_{\Delta x \to 0} \frac{u(x_0 + \Delta x, y_0) - u(x_0, y_0)}{\Delta x}. \quad (3)$$

For any continuous $u(x,y)$, given small $\Delta x$, Equation (3) should give a reasonable
estimate. Via Taylor series expansion, \( u(x_0 + \Delta x, y_0) \) can be expanded to

\[
\frac{\partial u}{\partial x}_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + O(\Delta x),
\]

where \( O(\Delta x) \) is the truncation error from truncating the Taylor polynomial [2]. This is the forward space-scheme. There are multiple ways to discretize any continuous function, two of which are important in the scope of this work, namely backwards and central space-schemes. The former comes from simply rearranging the terms:

\[
\frac{\partial u}{\partial x}_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x} + O(\Delta x). \tag{5}
\]

The central difference method subtracts the forward scheme from the backwards scheme:

\[
\frac{\partial u}{\partial x}_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + O(\Delta x)^2. \tag{6}
\]

The second derivative can be acquired using Taylor series expansions [2], which results in

\[
\frac{\partial^2 u}{\partial x^2}_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + O(x)^2. \tag{7}
\]

When the state of the system is changing with respect to time, time also needs to be discretized. A simple example of transient heat transfer is unsteady conduction, where the past states are required to solve the present state. Applying Equation (2) explicitly for temperature can be used to model unsteady conduction [2]:

\[
\frac{\partial T}{\partial t} = \alpha \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right), \tag{8}
\]

which can be discretized explicitly in the following manner [2]

\[
\frac{T_{i,j,k}^{n+1} - T_{i,j,k}^n}{\Delta t} = \alpha \left( \frac{T_{i+1,j,k}^n - 2T_{i,j,k}^n + T_{i-1,j,k}^n}{(\Delta x)^2} \right. \\
+ \frac{T_{i,j+1,k}^n - 2T_{i,j,k}^n + T_{i,j-1,k}^n}{(\Delta y)^2} \\
+ \left. \frac{T_{i,j,k+1}^n - 2T_{i,j,k}^n + T_{i,j,k-1}^n}{(\Delta z)^2} \right). \tag{9}
\]

The above discretization essentially considers the neighboring nodes of node that is being solved in their respective dimension (x, y or z). This system is therefore central with respect to space, as it considers nodes from both sides of the node being solved, and forward with respect to time.
2.2.1 The enthalpy method

The enthalpy method allows us to further simplify the heat equation discretization [10, 11]. The equations presented in the earlier sections only apply if the heat conduction $k$ is assumed to be constant and independent of temperature. Heat conductivity however is dependent on temperature, and in an unsteady conduction system it is then also dependent on the space as the temperature within any given material is also a function of space $(x, y, z)$. Writing Equation (8) with $\alpha = k/\rho c_p$ expanded out and latent heat $S$ added to the equation then yields

$$\rho c_p \frac{\partial T}{\partial t} + S = \nabla \cdot (k \nabla T).$$

(10)

The right side of the equation can further be simplified by performing a Kirchoff transformation to heat conductivity. The result is a Kirchhoff function that is solely dependent on the temperature:

$$K(T) = \int_{T_0}^{T} \kappa(\xi) d\xi,$$

(11)

which can be used to rewrite (10) as

$$\frac{\partial H(T)}{\partial t} = \Delta K(T).$$

(12)

Discretization of the enthalpy term then becomes simple:

$$\frac{\partial H(T_{i,j,k}^d)}{\partial t} \approx \frac{H_{i,j,k}^d - H_{i,j,k}^{d-1}}{\Delta t_d}.$$

(13)

2.2.2 Explicit methods

Explicit scheme, or forward time central space (FTCS) produces a time-explicit equation for temperature. Using Equation (8) and Equation (9) using the knowledge that initial distribution of temperature is available, the discretization of Equation (8) can be written as [2]

$$T_i^{n+1} = r(T_{i+1}^n + T_{i-1}^n) + (1 - 2r)T_i^n,$$

(14)

where $r$ is defined as $\alpha \Delta t/(\Delta x)^2$. As the equation is defined explicitly by the last time step, all nodes $0, \ldots, i$ can be solved simply by using Equation (14) and using values from the surrounding nodes on their last time step. This makes parallelizing explicit schemes very simple: in any given time step there are $i$ independent equations that need to be solved, and the time taken for solution is always known beforehand, as the magnitude of time step needs to be defined before starting computing, and it can be precisely calculated how long it takes to compute each time step, as it
always contains the same number of arithmetic operations. The error term of explicit scheme is linear with respect to space and quadratic with respect to time [2]:

\[ O(t) + O(x^2). \]  \hspace{1cm} (15)

The downfall of explicit method is that the time steps have to be sufficiently small in order for the result to converge at all [2]. The stability constraint can be expressed as

\[ \frac{\alpha \Delta t}{(\Delta x)^2} \leq \frac{1}{2}. \]  \hspace{1cm} (16)

Exceeding the stability constraint can result in physical impossibilities, such as heat flowing from colder region to warmer region. Compared to implicit scheme, picking a time step distance is harder and often using explicit scheme results in slower computation times due to increased amount of time steps, even though computing a single time step with the explicit scheme is much faster than implicit scheme, as there is no system of linear equations that needs to be solved.

The explicit scheme is visualized in Figure 3, where \( i \) is the finite space step and \( j \) is the finite time step. The figure illustrates that only previous values are required to find the solution for next time step.

![Figure 3: Explicit method visualized. The space step is represented by \( i \), while \( j \) represents the step in time. The system solves the filled red node \((i, j + 1)\).](image)
2.2.3 Implicit methods

Implicit methods, or backward time centered space (BTCS), use data from last time step and compute multiple values from the current time step simultaneously. Implicit methods are usually inherently stable. This means that the parameters for any given finite difference method model using implicit method can be chosen purely on the basis of desired accuracy [2]. The drawback is that finding each new step in any given iteration requires solving a large system of linear equations, unlike in explicit method where the next time step can easily be calculated from the previous time step.

Implicit methods introduce a time weighing factor, \( \theta \), which is 1 for the simple implicit scheme and \( 1/2 \) for the Crank–Nicolson method. The discretization is then done as follows: [2, 12]

\[
\frac{T_i^{n+1} - T_i^n}{\Delta t} = \frac{\alpha}{(\Delta x)^2} \left( \theta(T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}) + (1 - \theta)(T_{i+1}^n - 2T_i^n + T_{i-1}^n) \right). \tag{17}
\]

Using \( \theta < 1/2 \), the method has the same stability constraints in place as the explicit method, however when using \( \theta \geq 1/2 \), the equation is always stable. When \( \theta = 1 \), the error term is same as it is in the explicit method. Grouping by the factors, Equation 17 can be re-written as [2]

\[
b_i T_{i-1}^{n+1} + d_i T_i^{n+1} + a_i T_{i+1}^{n+1} = c_i, \tag{18}
\]

where \( a, b, d \) contain the constants and \( c \) contains the information from the previous time step. In the case of traditional implicit scheme, \( c \) only contains the node \( i, n \) as \( \theta = 1 \) and the term containing the space information from previous time step is eliminated. The visualization of simple implicit scheme can be seen in Figure 4. The figure shows, \( i, j + 1 \) is dependent on \( i - 1, j + 1 \) and \( i + 1, j + 1 \) and they are also dependent on \( i, j + 1 \), which is the inherent reason why a system of linear equations needs to be solved.

As the system of equations (18) contains multiple unknowns \((T_{i-1}^{n+1}, T_i^{n+1}, \text{ and } T_{i+1}^{n+1})\), it must be solved. A common method to solve system of linear equations is to solve them numerically, which means that multiple iterations per time step are required. The iterations are continued until either the result converges, or iteration limit is exceeded. If iteration limit is exceeded, the calculation is usually considered a failure and is halted. Iteration limit is usually selected so that it is magnitudes larger than the average case. Solving (18) can be parallelized but it is not as straightforward as parallelizing the explicit method. Parallelizing and solving the equation is explained more thoroughly later on in this thesis.
2.2.4 Crank–Nicolson method

Crank–Nicolson method can be thought of as central time, central space (CTCS) or a hybrid method between the FTCS and BTCS methods. It uses information both from current and the previous time step. Crank–Nicolson tends to be faster than the normal implicit backwards method, but slower than the explicit method per time step, as Crank–Nicolson method still requires solving a system of linear equations at all time steps. Crank–Nicolson scheme utilizes (17) with $\theta$ value $\frac{1}{2}$. The visualization of Crank–Nicolson scheme can be seen in Figure 5.

Crank–Nicolson generally converges faster than the simple implicit scheme, but it can contain some oscillation if the space difference $\Delta x$ is large [2, 12]. The error term of Crank–Nicolson is quadratic both with respect to time and space:

$$O(t^2) + O(x^2).$$  \hfill (19)

The faster convergence is explained by the larger truncation error; larger truncation error is directly rated to rate of convergence [13]. Any potential oscillation is also explained by the larger truncation error.
Figure 5: Crank–Nicolson method visualized. The space step is represented by $i$, while $j$ represents the step in time. The system solves the filled red node $(i, j + 1)$.

Crank–Nicolson method is the one used in this work due to the presumed faster rate of convergence. The practical implementation of Crank–Nicolson scheme is presented later on in this work.

2.3 Iterative methods: Jacobi and modified Gauss–Seidel method

As has been shown in the earlier chapters, purely implicit discretization methods and the Crank–Nicolson method results in a system of linear equations. Consider a simple linear system:

$$Ax = b,$$  

(20)

where $A$ is a $n \times n$ matrix, while $b$ and $x$ are $n$-dimensional vectors. Both iterative methods, the Jacobi [14] method and the modified Gauss-Seidel [14] method start by making an initial guess for the unknown variables $x_1, \ldots, x_n$. The linear system can then be expanded and written out for three variables to illustrate solving one iterative step with either method:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2,$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$  

(21)
which we can then trivially solve for $x_1$, $x_2$ and $x_3$ with respect to the unknown variables. To solve the system using the Jacobi method, each of the rows of the system of linear equations is solved by making an initial guess for all of the unknown variables $x_1, \ldots, x_n$ at once. Each row can be solved independently of any other row and thus can be entirely parallelized trivially. The whole system is then solved using the initial guesses. After each solution, the solutions from the previous iterations are used in place of their respective $x_1, \ldots, x_n$ and the iteration is continued until the solutions begin to converge.

The Gauss–Seidel method in contrast to the Jacobi method uses values for $x_1, \ldots, x_n$ as soon as they become available. Let us consider Equation (21): the Gauss-Seidel iterative scheme would within the first iteration use initial guesses for $x_2$ and $x_3$ to solve for $x_1$. However, when solving for $x_2$, the Gauss–Seidel method would use the newly computed value of $x_1$ instead of the initial guess for that value. This makes Gauss–Seidel inherently sequential; each row except for the first one is dependent on the solutions of the previous rows.

The modified Gauss-Seidel scheme used by some variations of the model in this work parallelize and modify Gauss-Seidel in a manner such that each independent core solves for one row of the linear system, and then any other independent core can utilize that solution as soon as it becomes available: the modified Gauss–Seidel is not bound by order. This however requires taking care that writing and reading of the values is either atomic or that there is a system in place that can broadcast only values that have been computed and fully written to their respective memory addresses.

2.4 Parallel computing

From 1970s to early 2000s, the computing power was largely increased by increasing clock speeds of processors and by improving microprocessor architecture. In the past, major advancements in computing power have included instruction level parallelism, increasing clock speeds, cache improvements and branch prediction among other techniques [15]. Recently, however, the improvements have come from adding multiple CPU cores rather than increasing clock speeds, and even more recently, utilizing GPUs in general purpose computing [16]. To fully utilize and adapt to the new hardware, software must be written in a way that takes advantage of the parallel computing possibilities.

Modern scientific models however are becoming increasingly more demanding in terms of computation resources; either they have become more precise by increasing problem size or they have become more time critical. The model used in this particular thesis for example has the end goal of being used in real time process control. Real time process control sets some limits that traditional models do not have: It has to be reliable, it has to always compute the same results within the same bounded time or it will fall behind the process itself. In addition, it is not the
only model used in the system but other models will be basing their results on this model; therefore it is even more critical that the computation of the temperature distribution is as fast as possible while maintaining accuracy. Parallel computing is one possible solution to this issue; to fully utilize modern hardware, software must take full advantage of the parallel processing capabilities of the hardware. Many of the algorithms used in modeling have been written with sequential processor in mind and thus some of those algorithms might be inherently sequential, where the order of computation matters, or contain portions that are sequential.

Most problems do not parallelize entirely and therefore increasing the amount of independent cores does not necessarily scale linearly with the speedup. Amdahl’s law can be used to determine the theoretical output of any given algorithm after parallelization. Amdahl’s law [17] splits algorithms to two parts: Serial and parallelization portion. The maximum benefit from parallelization is defined as

\[
speedup = \frac{1}{R_s + \frac{R_p}{n}} \tag{22}
\]

where \( R_s \) is the ratio of sequential portion in the algorithm, \( R_p \) is the ratio of the parallel portion and \( n \) is the number of cores. For example, if in a given algorithm there is a 5% portion that is sequential, the speedup given by Amdahl’s law is \( 20 \times \) as the amount of cores approaches infinity. Amdahl’s law in general can be considered valid when problem size stays constant relative to the sequential implementation. Increasing problem size however more often than not, does not increase the sequential part linearly but rather the constant term tends to grow slower than the more intensive, usually parallel computation term: increasing problem size tends to increase the parallel-sequential ratio [18, 19]. To put it in other words, according to some more modern takes on Amdahl’s law, growing problem size favors parallelism. For example, if the sequential portion of a problem has time complexity of \( O(n) \), and is currently taking a certain portion of computation time, and the parallel portion has time complexity of \( O(n^3) \), then by increasing \( n \), the relative portion of the inherently sequential part of the algorithm becomes smaller relative to the parallel portion.

As has been mentioned earlier in this work, the original purpose of GPUs – to process graphics has shifted more towards general purpose usage; it is now easier for a programmer to give explicit instructions to GPUs, and they are no longer limited to having a texture mapping units that can only compute textures and data no longer needs to be broken down into e.g. textures or 3/4D-vectors that are native to GPU, but rather they can accept almost all data types now [20]. This in turn has opened the field for all kinds of models to utilize GPUs to either accelerate a specific part of the model using existing pre-built libraries such as cuBLAS [21], clBLAS [22] that contain standard basic algebra subroutines. Alternatively a programmer can now write parts of the model entirely for the GPU using either CUDA [20], which allows to write fully general purpose code for graphical processing units produced by NVIDIA or OpenCL [23], which supports a large variety of hardware from Intel CPUs to AMD CPUs and from NVIDIA GPUs to AMD GPUs.
As established earlier, the commodity hardware used is not getting faster via increasing clock speeds, the computation speed is now improving by increasing the throughput via increased amount of independent cores that can complete tasks in parallel [24, 3]. However, parallelizing software so that it can utilize hardware of one computer might not be enough in the future, scaling out from using just one computer to using clusters of computers can be done to further increase the throughput, albeit usually at cost of increased latency [25]. Cloud based cluster computing can become especially troublesome with large datasets; suddenly there is a new issue of how is the data distributed across the nodes. This in turn, leads to many new questions; How to optimize the data storage so that each node contains the data it needs for its computations so it does not have to specifically query for it? How much of that data is kept in DRAM memory? How to deal with a single node (computer or a server rack in some cases) failing? Such issues make it difficult to utilize cluster computing for a model that is supposed to find results always in bounded time. There are multiple solutions that solve some of these problems, such as Apache Spark [26] and Hadoop MapReduce, which is an open source implementation of Google MapReduce [27]. Due to ever increasing demands from scientific models, but diminishingly improving hardware, it is entirely possible that at some point parallelizing across the same computer will not be enough to acquire suitable results and that the computation will have to be distributed across multiple computers. Most of these solutions are very high latency solutions compared to anything one could do on a single computer.

2.4.1 Parallelization in modeling

Ordinary differential equations or partial differential equations describe a large variety of physical and thermodynamical problems [28, 29]. As established earlier in this work, finding analytical solutions to those equations can prove troublesome, but they can then be discretized and solved iteratively. As has been established earlier, solving massive sets of linear equations is computationally very intensive. The independent nature of matrix operations is ideal case for parallelization: a massive problem can be turned into many very small problems. Solving many small problems is an ideal use case for GPUs that have hundreds or thousands of relatively slow cores compared to CPUs that tend to have few more powerful cores [30].

In general when writing code for GPU it does not make sense to compute everything on the device (GPU), due to various factors: it takes time to transfer memory from the host (CPU) to the device and not all problems parallelize entirely or at all and GPUs do not perform well on sequential tasks compared to CPUs. Due to the limitations and constraints, the code that is to be optimized is inspected closely and usually only the most computationally intensive parts are written for the GPU using e.g. CUDA or OpenCL. That is to say that when general purpose computing on graphics processing units (GPGPU) solutions are discussed in this thesis, it refers to implementing specific parts on the GPU. There is always a host code (CPU) implementation that takes care of non-critical parts of the program. For example,
Klimes et al. wrote two short kernels for their entire model, one that computes the time steps and one that updates the boundary conditions [31].

There is a plethora of various research and industrial projects where parallelization either on CPU or on GPU have been utilized for solving FDM-schemes either implicitly or explicitly [29, 31, 32, 33]. Meshless methods have been parallelized as well, Singh et al. were able to speed up their meshless EFG model by $4.6\times$ using eight cores [34]. Klimes et al. in turn used CUDA in their continuous casting model with speedups up to $30 - 68\times$ [31]. They utilized an explicit numerical scheme for discretization. They further elaborated that they chose explicit scheme as it does not require sequential processing in nested loops. However, it should be noted that it is entirely possible to make similar implicit method where the only inherently sequential parts are the iterations to solve one time step and that each one of those iterations can be entirely parallelized and has already been done previously [32]. In general it would seem that explicit methods are more common in the literature, and they seem to state that parallelization of those explicit versions is easier than implicit versions or that the explicit versions somehow allow for greater degree of parallelization. The speedups in the literature seem to range from barely improving at all, to almost $100\times$ improvements. Something to take into consideration is the portion of truly parallel instructions and problem size.

Micikevicius et al. showed that some FDM problems can be scaled linearly across multiple GPUs [30]. That is to say that it is possible that in the future these kinds of models could be scaled out to multiple GPUs on the same computer for time critical applications and possibly even across multiple computers or server racks if latency is non-critical. For example Wheatstone et al. have shown interest to cluster based GPGPU computing for their advanced regional prediction system [35]. All in all, it would seem that parallel computing has become widespread both in the research and in the industrial circles.
3 Research material and methods

The basis of this work is a pre-existing transient heat transfer model, produced by Casim Consulting Oy [36]. The model is based on the enthalpy method presented earlier in this work and the numerical scheme is implicit (Crank–Nicolson). This model is used as a baseline in the benchmarks and the aim of this work is to speed up the computation time of this specific model.

For this purpose the algorithms used in this heat transfer model are inspected for any parallelization possibilities and evaluated for their efficiency. All means that are practical and reasonable with respect to the time allocated to this thesis are taken to optimize the model: inspecting and optimizing the data structures to optimize memory access patterns, optimizing pre-processing and parallelizing the code both on CPU and GPU.

The core method of writing code for this thesis has been very test driven: the first step of this work was to produce datasets using the baseline model, which are used for validation purposes. After each change to the code, the tests were ran to make sure that the model always converges to the same numerical values within the same time steps within certain error $\Delta$ term, that is caused both by floating number precision and by the convergence constraint $\epsilon$. A smaller convergence constraint leads to improved precision and increased number of iterations per time step.

3.1 The slab furnace model

Temperature depends both on space $(x,y,z)$ and on time $t$. Function of temperature within the slab can therefore be expressed as

$$T = T(x,y,z,t).$$

(23)

Using (10) we can define the heat transfer equation as a piecewise linear function of enthalpy:

$$\frac{\partial H(T)}{\partial t} = \Delta K(T).$$

(24)

The boundary conditions can be expressed as

$$-\nabla K(T) \cdot \vec{n} = h(T - T_{ext1}) + \sigma \epsilon (T^4 - T_{ext2}^4),$$

(25)

where $\vec{n}$ is the external unit vector of the slab, $\sigma$ is the Stefan-Boltzmann constant, $\epsilon$ is the emissivity, $h$ is the heat transfer coefficient, $T_{ext1}$ is the known temperature for convection at slab boundary and $T_{ext2}$ is the known temperature for radiation at furnace wall [37]. Both $T_{ext1}$ and $T_{ext2}$ are data points that are given by the process itself on each time step. There are temperature sensors that measure the temperature of the furnace wall and the temperature of the atmosphere within the furnace. The geometry of the model with the positions of $T_{ext1}$ and $T_{ext2}$ can be seen in Figure 6.
Additionally an initial guess for the temperature vector $T$ at $t = 0$ is provided below.

The temperature function can then be written with respect to space and time. Additionally an initial guess for the temperature vector $T$ at $t = 0$ is provided for the model

$$T(x, y, z, 0) = T_{\text{init}}(x, y, z).$$

(26)

The temperature distribution is based on prior knowledge; it is common for the slabs to either let the slabs cool down to room temperature so that their quality can be verified before they go into the reheating furnace, or they are kept in a controlled temperature (e.g. in a furnace) before going into the reheating furnace. In either case, the initial distribution given to the model is usually relatively close to reality and easy to obtain. More detailed image of the slab can be seen in Figure 7.
being the exact solution, we denote the approximate result acquired from the finite difference method as \( T^d \). Entalphy-term can then be approximated

\[
\frac{\partial H(T^d_{i,j,k})}{\partial t} \approx \frac{H^d_{i,j,k} - H^{d-1}_{i,j,k}}{\Delta t^d}.
\]

(27)

Laplace term can be obtained by taking first and second order partial derivatives by \( x, y \), and \( z \) of Equation 24, which yields

\[
\Delta K(T^d_{i,j,k}) = K_{xx}^{i,j,k} + K_{yy}^{i,j,k} + K_{zz}^{i,j,k}.
\]

(28)

The solution is then discretized using Crank–Nicholson method and the resulting system of linear equations solved using modified Gauss–Seidel–Newton–Raphson method (for Newton–Raphson method, see e.g. [38, 39]). For parallelized version a Jacobi-version is utilized.

The resulting system of linear equations becomes [36]

\[
M^d H(T^d) - AK(T^d) - BT^d - C(T^d) = F,
\]

(29)

where \( M \) is the inverse of time step multiplied by the inverse matrix, Where \( A, B \) and \( C \) contain the factors from \( \Delta K(T^d_{i,j,k})^d, T^d \) and \( (T^d)^4 \) respectively. Additionally, \( F^0 \) is defined as the constant vector of the Laplace term of (28) as follows:

\[
F = F^0 + M^d H(T^{d-1}) + AK(T^{d-1}) + BT^{d-1} + C(T^{d-1})^4.
\]

(30)

The equation can then be produced to solve for a specific iteration step \( d \). If using Gauss–Seidel method, the equation can use solutions from either the current or the previous iterations, while Jacobi only uses the solutions from previous round. Solution for specific iteration step \( d \) can then be found [36]

\[
M_{i,i} H(T^d_i) - AK(T^d_i) - B_{i,i} T^d_i - C_{i,i} (T^d_i)^4 = F_i + \sum_{j=0,j\neq i}^{6} A_{i,j} K(T^d_{i,j}).
\]

(31)

We then define the right hand side of the equation as \( \sigma \):

\[
\sigma = F_i + \sum_{j=0,j\neq i}^{6} A_{i,j} K(T^d_{i,j}),
\]

(32)

where \( i \) is the node being currently solved and \( j \) denotes the neighboring nodes. Each neighboring node is given value from 0 to 6, and the node itself is always given the middle spot, \( j = 3 \), which is ignored when computing the node itself. \( K(T) \), \( H(T) \) are known beforehand; they can be used to compute the constant vector \( F \) and therefore the right-hand side of the equation is known. Using the fact that \( K(T) \) and \( H(T) \) are piecewise linear and denoting \( C \) as \( \alpha \), all other factors of \( T^d \) as \( \beta \) and other factors as \( \gamma \) we can then define a piecewise linear \( \psi \) function and re-write Equation 31 as

\[
\psi_i(T, m) = \alpha_i T^4 + \beta_{i,m} T + \gamma_{i,m},
\]

(33)
Or to simplify further, using the $\sigma$ definitions from (31) and (32) it can be written as

$$\psi_i(T, m) = \sigma,$$

where $i$ is the node being solved and $m$ the temperature interval index. The index can be solved by iterating through Equation 34, using an initial guess of index $m$, and then incrementing it if the value is too low and decreasing it if the value is too high. The $\sigma$ can be computed before and therefore only the index and temperature remain unknown. This equation is analogous with the Equation (18) presented earlier alongside other discretization methods. The unknown index $m$ can be solved before solving for the exact temperature through an iterative algorithm. The algorithm moves forward or backwards through the piecewise linear $\psi$ function, until the computed $\sigma$ value is within a known interval. Figure 8 further illustrates the method for finding the temperature interval.

![Figure 8: Illustration for the algorithm that finds the reference temperature interval in the piecewise linear $\Psi(T_d, m)$ function, which is dependent on material properties as defined by $K(T)$ and $H(T)$.](image)

After the index is known, $T^{(d)}$ can be solved with Newton–Raphson method. More specifically, one iteration step completes one Gauss–Seidel–Newton–Raphson or
Jacobi–Newton–Raphson iteration iteration step towards the solution of the system:

\[ T_i^d \approx T_i^{d-1} - \frac{\psi(T_i^{(d-1)}, m) - \sigma_i}{\psi'(T_i^{(d-1)}, m)}. \] (35)

Using (33) then further yields the final equation for temperature:

\[ T_i^d = \frac{3\alpha_i(T_i^{(d-1)})^4 - \gamma_{i,m} + \sigma_i}{4\alpha_i(T_i^{(d-1)})^3 + \beta_{i,m}}, \] (36)

where \( m \) is the index that has been solved. The ending criteria for the calculation is simply the convergence of the iteration. The convergence can then be checked trivially:

\[ \frac{T_i^{(d)} - T_i^{(d-1)}}{T_i^{(d)}} < \epsilon, \] (37)

where Equation (37) must be true for all \( i \). If even one node does not converge, a new iteration step is required. Iteration is continued until a solution is found or iteration limit is met. If iteration limit is exceeded, the calculation stops: this is considered a critical failure in the system. As soon as convergence is found, next time step is initiated.

The time steps themselves are split into two parts. One is the time step given by the underlying process. In this case, the furnace pushes temperature readings every thirty seconds. This thirty second interval is then broken into multiple smaller time steps internally. Each time step is solved independently and as soon as last of the internal time steps has been completed, the process is given new temperature readings from the model (the temperature vectors resulting from internal time steps are not transferred from the solver class).

### 3.2 The CUDA programming model

There is a large discrepancy between the computing capability of GPU and CPU when it comes to measuring out the raw floating point operations per second [40]. This is due to inherent differences in processor design. Modern consumer-grade CPUs have 2-8 cores, while a mid-range GPU can have hundreds of cores [41]. However it should be taken into consideration that many floating point operations can be vectorized on CPUs, allowing the CPU to process multiple floating point operations within a single instruction. CUDA is a parallel programming platform and a programing model created by NVIDIA that allows programmer to utilize NVIDIA GPUs for general purpose computing [40].

CUDA programs are usually written in a manner such that only one set of instructions is written but it is performed on multiple data points. Operations are only written once, per node or per multiple nodes in case striding of some sort is utilized. Unique indexes are then derived from grid, block and thread identifiers. For example,
instead of writing a for loop that performs $N$ operations, a number of threads equal to $N$ are instead launched, each of which will perform the instructions that would have been within the loop.

Indexing in CUDA is hierarchical: it is divided into grids, blocks and threads. Grids are the largest unit of them all, followed by blocks and finally threads. Grids consist of $N$ blocks that, in turn, consist of $M$ threads. Each thread has a globally unique index that can be computed from unique block index (position of the block within a grid) and thread index (position of the thread within a block). Blocks can be one, two or three dimensional. Figure 9 shows a two-dimensional 3x3 grid and Figure 10 shows a 3x3 block. The dimensions of blocks and grids are usually determined by the size of input data.

![Figure 9: A 3x3 CUDA grid](image)

![Figure 10: A 3x3 CUDA block](image)

CUDA programs rely heavily on mapping the threads to data. For this reason, the CUDA API contains some essential keywords that can be used by the thread that is physically executing the kernel to retrieve the global index of that thread. To uniquely access any thread with a grid in one dimensional system, three pieces of
information are required: the unique index of the given block, the dimension of the blocks within the grid and finally the thread index within its respective block. The index can then be retrieved trivially:

$$\text{idx} = \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x},$$

where blockIdx, blockDim and threadIdx are keywords provided by CUDA API that contain the relevant index information. For example, in a one dimensional grid, where grid width is 3 blocks, and block width is 3 threads, there are total of 9 threads which can all be accessed on an interval where blockIdx.x and threadIdx.x can have values in the range $[0, \ldots, \text{blockDim.x} - 1]$.

Expanding to two dimensions is similarly quite easy. Usually the reason to use two, or three dimensional blocks is dimensionality of the data, so it makes sense to acquire both x and y indexes:

$$i = \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x}$$
$$j = \text{blockIdx.y} \cdot \text{blockDim.y} + \text{threadIdx.y},$$

where the pair $(i, j)$ includes all combinations that the grid-block system allows for. Additionally, a system of 2D blocks and 2D grids can also be accessed with single globally unique index, that can be seen below:

$$\text{idx} = (\text{blockIdx.x} + \text{blockIdx.y} \cdot \text{gridDim.x}) \cdot (\text{blockDim.x} \cdot \text{blockDim.y}) + (\text{threadIdx.y} \cdot \text{blockDim.x}) + \text{threadIdx.x}.$$  

In general, when launching a CUDA kernel, the grid size is determined from the input size. This ensures that the grid contains enough threads for each data point. To make sure that the kernel does not exceed the data boundaries, the kernel is usually provided with data dimension as an argument to check that the thread index is within the desired range. The grid size can be computed as

$$\text{BlocksPerGrid} = \frac{N + \text{ThreadsPerBlock} - 1}{\text{ThreadsPerBlock}},$$

where $\text{ThreadsPerBlock}$ is usually 128, 256, 512 or 1024, and $N$ is number of nodes. Should the grid dimension ($\text{BlocksPerGrid}$) ever amount to a number greater than $2^{31} - 1$, which is the maximum grid x-dimensionality that CUDA 8.0 for devices of all compute capability provides [40], striding mechanisms will be utilized in the kernel itself.

### 3.3 Validation

To properly assess whether the CUDA and improved CPU implementations produce the correct results, test data was recorded for varying slabs using simulation time of
120 minutes with the time step being 30 seconds, using a single threaded solution.
For each time step, temperature was recorded and saved at the core of the slab,
corners of the slab, and sides of the slab. The produced data was then serialized and
saved in JSON format. Running tests compares the relative difference of computed
values against the recorded values. The exact model was then ran against the data
to assess how closely the results compare in subsequent runs while using double
precision floating points. The maximum allowed relative error was chosen to be
$10^{-7}$, as that was the lowest number producing accepted tests with the same model
that the data was recorded from. Using $10^{-8}$ as maximum relative error produced
approximately one failed node (out of six) per time step per slab, while $10^{-9}$ caused
almost all of the test case nodes to fail.

The validation data was acquired from a single threaded version utilizing Gauss-Seidel
scheme. After running the same model with Jacobi-scheme, it was verified that both
methods converge to the same results as they should. Validation was constantly used
during the process of building the model. After each change, either to the kernel or
to the host code, tests were ran to make sure that the model is still functioning as it
should.

### 3.4 Evaluation of the present implementation of the model

Memory layout was one of the first items that was noted to be likely detrimental
to the performance. The baseline version used indexing arrays that used value
of one array to access value of other array in the manner of $a[b[i]]$: these were
replaced with direct indexing where possible. Another issue was two dimensional
arrays (e.g. $a[y][x]$): they could not be easily used in CUDA as is and thus had
to flattened. Another issue with the two dimensional arrays was that they were
formatted in a way that essentially guaranteed that no memory accesses could be
sequential. The memory layout for e.g. the Kirchhoff or Enthalpy function can be
seen in Figure 11. As Kirchhoff values are always acquired in pairs by each thread,
the second access would always be guaranteed to be $nx$ memory slots away from
the first one, essentially guaranteeing that no memory accesses are sequential. To
fix this, the matrix was transposed and flattened, as can be seen in the Figure 12.
The baseline implementation contains all of its arrays except for the temperature
vector in this or very similar format. All of those were transposed and flattened in
the manner that the figures show, which guarantees at least some level of sequential
memory access. Both figures show in the flattened array, a typical memory access
task that two independent threads A and E would do as they fetch a pair of either
Kirchoff or enthalpy function values from the respective array.

There were a number of other indexing related performance bottlenecks as well. As
mentioned earlier, a lot of the indexing was indirect, as the indexes for e.g. the
constant matrix $A$ were pre-computed in the preprocessing and then given to solver
class as parameters. These included computation of neighbor vectors, which for any
given node in 3D space is simply $n + 1$ and $n - 1$ in $x$ direction, $n + nx$ and $n - nx$
in $y$ direction and $n + (nx \times ny)$ and $n - (nx \times ny)$ in $z$-direction. As according the profiler results, full utilization of the GPU was not close to being optimal, computing direct indexes on the fly as opposed to pre-computed index tables was determined to be the better choice, as fetching items from global GPU memory can be slow. This also greatly reduced the amount of memory that needed to be transfered to GPU, further streamlining the process, and resulted in cutting of the kernel initialization time in half.

Additionally, it was noted very soon in the benchmarks that a lot of time is spent in setting the boundary conditions after each time step. This problem was reduced by parallelizing the boundary condition computation in the slab model class. The overhead of constant time taken by the boundary condition computation was benchmarked and noted by the writer of this thesis, but implemented by Casim Consulting Oy, as it was part of the program that the writer of this thesis did not have direct access onto. Casim Consulting Oy also provided the experimental CLI/C++ version
of the model.

In Table 1 we can see the baseline performance breakdown for 0.56 million (A) and 1.11 million (B) nodes. Here getKirchoff corresponds to Equation (32), calcIterCore is the iterative process that is done multiple times on each time step for all core points, calcIterBound is the same process for boundary points and getPsiValue corresponds to Equation (33). As can be seen in the table, the overhead of calling the solver is relatively huge in the A-version, while it gets reduced by over ten percentage points after doubling the nodes in version B. Version B clearly spends more time computing than anything else. From that we can conclude that at least a portion of the sequential code (external code, calling HeatSlabSolver) can be reduced by increasing the problem size, which should give more favorable results according to Amdahl’s law, as the portion of parallel code increases.

Table 1: Performance breakdown, baseline

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU cycles spent (%) A</th>
<th>CPU cycles spent (%) B</th>
</tr>
</thead>
<tbody>
<tr>
<td>getKirchhoff</td>
<td>21.86</td>
<td>28.28</td>
</tr>
<tr>
<td>calcIterCore</td>
<td>21.19</td>
<td>27.45</td>
</tr>
<tr>
<td>HeatSlabSolver::run</td>
<td>13.86</td>
<td>1.18</td>
</tr>
<tr>
<td>external code</td>
<td>10.74</td>
<td>3.33</td>
</tr>
<tr>
<td>calcIterBound</td>
<td>7.77</td>
<td>3.75</td>
</tr>
<tr>
<td>getPsiValue</td>
<td>6.23</td>
<td>14.63</td>
</tr>
<tr>
<td>calcConstantCore</td>
<td>4.48</td>
<td>3.37</td>
</tr>
</tbody>
</table>

3.5 Analysis on parallelization possibilities of the existing model

Each iteration can be independently parallelized entirely but two time steps can never be parallelized, as one time step depends entirely on the preceding time step. In any given iteration, $m$ operations will be done for each of the $n$ nodes. This allows hundreds of CUDA cores to work on the same iteration at the same time. However, it also means that once almost all of the threads are finished, but there are some threads still lingering, the entire process has to wait for those last threads to finish before moving to the next time step.

The boundary conditions can be parallelized, as all it involves is filling constant vectors with constant data that has been given from the process. Initialization of all of the other data structures can also be parallelized as much as the memory bandwidth allows for.

Technically the current Gauss–Seidel method that is used in the single threaded baseline version can be parallelized, if the underlying system can guarantee that each write to memory is atomic and that each read can only read the previous
or the current value. This is something that e.g. the common language interface runtime specifications \[42\] guarantee for and up to native integer size (64 bit for 64-bit systems and 32 bit for 32-bit systems). CUDA, however provides no such guarantee and therefore the resulting data race could produce unpredictable results. To overcome this, one possibility is to simply switch to Jacobi-scheme for the CUDA-implementation. The added requirement of this method is reserving one additional temperature vector to hold values from the previous iteration. This way the algorithm can alternate between reading from the arrays, starting by reading from the initial values \(T\) to the new array \(T_1\). On the next iteration the data is read from \(T_1\) and written to \(T\). This prevents the same memory location being read or written at the same time.

3.5.1 Time complexity of current implementation

Currently it would seem that the time complexity of the current implementation scales quite linearly with increasing nodes. Roughly 100 operations are performed per node, including multiplications, divisions, comparisons and addition operations. Essentially the current time complexity is \(O(n)\) that has some constant term.

However, there are a lot of constant elements in the project that have a huge effect on the computation time. Furthermore, the constant time seems to grow constantly with growing problem size, and as the computation time here grows on a constant factor as well, increasing amount of processors may not yield as great improvements as one could expect.

3.5.2 Theoretical running times of parallelized implementation

NVIDIA does not directly report the FLOP/s that the device can perform. The device used in this case is quadro K2200. The manufacturer of the device however provides CUDA core number which is 640. \[41\] The clock frequency was determined using GPU-Z, which was 1.05 GHz. \[43\]

As any single core can perform floating point add and multiply (fused multiply add) per operation per cycle, we can compute that the peak instructions per cycle is 2 per core. \[44\] It should be noted that the current instructions per cycle in this code is closer to 1 than 2 as per nvprof-profiler.

We can compute the theoretical peak FLOP/s with the following formula:

\[
\text{(CUDA cores)} \cdot \text{(instructions per core)} \cdot \text{(GPU Frequency)}. \tag{42}
\]

Other constants can be seen in Table 1. Double FLOP/s value was computed by dividing single FLOP/s with 30.

A few assumptions were made to compute these results:
Table 2: Constants

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS (single)</td>
<td>1344</td>
<td>GFLOP/s</td>
</tr>
<tr>
<td>FLOPS (double)</td>
<td>44</td>
<td>GLOP/s</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>80</td>
<td>GB/s</td>
</tr>
<tr>
<td>CorePoints</td>
<td>444609</td>
<td>N</td>
</tr>
<tr>
<td>BoundaryPoints</td>
<td>112002</td>
<td>N</td>
</tr>
</tbody>
</table>

1. Each instruction can be performed fused multiply add (FMA)-operation and therefore the peak FLOP/s can be used.

2. Comparisons take the same time as a standard FLOP.

3. There are so few divisions that they can be added to other multiplications. (4 per point vs. 85 per point)

4. double FLOP rate is roughly single FLOP rate divided by 30

It is clear that these assumptions cannot strictly hold true in the practical implementation: it is expected that the practical implementation can not be as fast as the theoretical peak execution rate. Note that equation (43) includes both mul.f64 and mul.wide, Equation (43) contains all fma operations and Equation (45) contains div.f64 operations. The only part of the code that can change in runtime, which is finding the temperature interval by computing the reference temperature function was calculated using the maximum value found in the tests, which was 5.

\[
\text{multiplications} = 31 \cdot \text{corePoints} + \text{boundaryPoints} \cdot 54 \tag{43}
\]

\[
\text{additions} = (67 + 6) \cdot \text{corePoints} + (81 + 8) \cdot \text{boundaryPoints} \tag{44}
\]

\[
\text{divisions} = 2 \cdot \text{corePoints} + 2 \cdot \text{boundaryPoints} \tag{45}
\]

\[
\text{comparisons} = 8 \cdot \text{corePoints} + 14 \cdot \text{boundaryPoints} \tag{46}
\]

Adding all these together and multiplying with the average case iterations (i=30) yields 2.2 GFLOP. The total size of all memory read (including reading the temperature matrix six times) is 90 MB. Computing the time spent reading memory using the memory bandwidth value in Table 1 value yields 1.13 ms.

The average times using the computed GFLOP/s (including + 1.13 ms from reading memory) can be seen in Table 2. For reference, the current double implementation calculates the iterations (i=30) case in 140-180 ms when the kernel init and the constant boundary condition initialization time are subtracted.
Table 3: theoretical computation speed

<table>
<thead>
<tr>
<th>Type</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>52</td>
</tr>
<tr>
<td>Single</td>
<td>4</td>
</tr>
</tbody>
</table>

3.5.3 Host code (CPU) optimization

Host code has been optimized in various ways. There was a 100 ms constant time on each time step calculation that originated from computing the boundary conditions. Parallelizing that constant operation shortened constant time to 50 ms.

Improved memory access patterns. The original version used two dimensional arrays (jagged arrays in the .NET framework) that accessed indexes via two memory accesses array\([x][y]\). Changing those arrays to one dimensional format and directly accessing the via one memory access (array\([x + nx \times y]\)) sped up the computation by a factor of 2 in the actual heat transfer computation done on CPU side, and in addition it cut 20–30 ms away from the boundary condition computation. In addition changing indexing, some arrays were transposed to allow for sequential memory access. One such case was the Kirchhoff function array, which includes two values for each row. The original format was \(N \times 2\), while the new format is \(2 \times N\).

The original model was also writing results to disk in between time steps. All disk writing and reading was removed from the intensive parts.

3.5.4 GPU-kernel implementation

Two CUDA kernels were implemented. \textit{CalcConst} and \textit{calcIter}. The former is used once per iteration, and the latter is called as many times time step takes to converge. Note that the external time steps given from the process are divided into smaller time steps internally in the solver for stability reasons, the minimum iterations for any given set of time step values is \(\text{step}_{\text{ext}}/\text{step}_{\text{internal}}\). The minimum iterations can only ever be reached if single internal time step only takes one iteration to converge: in such case, the time steps could be too small and could potentially be increased.

Both kernels utilize 1D grids. The amount of blocks per grid is calculated in similar way in both kernels, using Equation (41). In this work, \text{threadsPerBlock} was chosen to be 1024, as it performed slightly better than the other block sizes.

Essentially the algorithm that the iterative GPU kernel does can be broken down to four parts: Solve \(\sigma\), find the temperature index \(m\) and use it to complete one Newton–Raphson iteration step to acquire the temperature and finally check for convergence. Each kernel completes one combined Jacobi–Newton–Raphson iteration step. To further clarify the process, flowchart of the kernel produced can be seen in
The CUDA pseudocode implementation for the core points can be seen below. The boundary point kernel is similar with additional checks to eliminate nonexistent points. The boundary kernel can be seen in Appendix A. Note that both of the kernels are presented as standalone kernels: in the actual implementation both boundary and core kernels are device functions, while the global kernel function is a trivial function that merely calls the boundary and core kernels and keeps track of order the $T$ and $T_1$ vectors.

```c
// load libraries and initialize constants

__device__
double getPsiValue(int intervalIndex) {
    return Equation 34($T_i$, intervalIndex);
}
```

Figure 13: The kernel flowchart.
double getSigma(int neighborIndex) {
    return Equation 32(neighborIndex);
}

__global__
void calcIterCore(params) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < endCore) {
        // initialize variables
        n = core_index[idx];

        // get sigma values from all six nodes
        // surrounding the current node
        // -z
        sigma += getSigma(n - nx*ny);
        // -y
        sigma += getSigma(n - nx);
        // -x
        sigma += getSigma(n - 1);
        // +x
        sigma += getSigma(n + 1);
        // +y
        sigma += getSigma(n + nx);
        // +z
        sigma += getSigma(n + nx*ny);

        while (getPsiValue(T_ind[n] - 1) > sigma) {
            T_ind[n]--;
        }

        while (getPsiValue(T_ind[n]) < sigma) {
            T_ind[n]++;
        }

    double gamma = Equations 31 and 34
    double beta = Equations 31 and 34

    // jacobi scheme: read results from old array
    // and then write to a new array
    // T1 will be given to kernel as T
    // on the next iteration as a starting point
    T1[n] = Equation 35(sigma, gamma, beta, T_ind[n]);
    diff = T[n] / T1[n];
    if (diff < eps1 || diff > eps2)
        isConverged = false;
    }
}
3.6 Benchmarking

The primary results presented were computed with a $10 \times 0.2 \times 2$ m steel slab, with node density being 0.02 m. There are total of 556611 nodes in the primary results. The size of the slab was chosen to be very close to the common slabs used in the industry and the node density was been chosen so that the results are accurate enough for process control. Additional tests were conducted with 0.56 million, 1.11 million, 2.22 million and 4.44 million nodes. All of the benchmarks were ran four times and then the average time of those four time steps was selected as the average time taken per time step. The external time step used was 30 seconds and the internal time step was 5 seconds for all benchmarks. The hardware used in all of the tests can be seen in Table 4.

Additionally to verify that increased nodes result in linearly increasing time, three additional benchmarks were ran: 20 m, 40 m and 80 m long slabs were used in these benchmarks. The idea behind increasing dimensions rather than node density was the keep the accuracy within a similar range, but the tests could have been done with an increased node density just as well. The increased node density yielded very similar results to those that utilized increased dimensions. There are four different implementations of the model that were used in the benchmarks. Their details can be seen in Table 5.

### Table 4: The hardware used in the benchmarks

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel(R) Xeon(R) Processor E3-1230 v5</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA Quadro K2200 (PNY)</td>
</tr>
<tr>
<td>Memory</td>
<td>16 GB DDR4-2134 / 1067 MHz</td>
</tr>
<tr>
<td>Mainboard</td>
<td>HP 802F (Skylake)</td>
</tr>
</tbody>
</table>

### Table 5: Different versions used in the benchmarks

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Platform</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>GPU implementation of the model</td>
<td>.NET 4.0 &amp; managed CUDA C 8.0</td>
<td>single and double</td>
</tr>
<tr>
<td>CPU (.NET)</td>
<td>Optimized version of baseline</td>
<td>.NET 4.0 C#</td>
<td>single and double</td>
</tr>
<tr>
<td>CPU (c++)</td>
<td>C++ implementation of the model</td>
<td>.NET 4.0 C++/CLI</td>
<td>single and double</td>
</tr>
<tr>
<td>Baseline</td>
<td>The original model, starting point</td>
<td>.NET 4.0 C#</td>
<td>double</td>
</tr>
</tbody>
</table>

The original baseline version uses modified Gauss-Seidel scheme, as does the optimized version of the baseline as does the C++ version. GPU version uses a Jacobi-scheme. All of the versions that utilize purely CPU are used only for benchmarking purposes.
4 Results

The main results can be seen in Figure 14 and Figure 15. The baseline is same for both models, and it was originally computed in double precision, using .NET implementation of the model. The line labeled CPU is then the optimized CPU version, GPU version is the float or double precision CUDA powered version, while CPU (C++) is a version where the .NET-based solver-class has been replaced with a C++ version. Constant is the time taken to compute boundary conditions before each time step, and kernel constant is the time taken to transfer the required arrays to GPU memory. It also includes any other potential kernel overhead not separately accounted for. The constant time taken for kernel (labeled kernel constant in the figures) only affects the GPU-implementation. X-axis is the number of external time steps, where each incrementation is 30 seconds in simulation time.

Figure 14: Average computation time per time step (single precision). The GPU implementation is $6 \times$ faster than the baseline, $3 \times$ faster than the NET CPU implementation and $2.2 \times$ faster than the C++ implementation.

The difference of clear drops on computation time are explained by reduced iterations per time step as the slab approaches equilibrium and therefore each time step has smaller difference to the one before. The x-axis difference of these drops between the CUDA and .NET versions are explained by slight difference in the algorithm used. The .NET-version uses modified Gauss–Seidel scheme and relies on ECMA.
Figure 15: Average computation time per time step (double precision). GPU is $4.8 \times$ faster than the baseline, $2 \times$ faster than the NET CPU implementation and $1.5 \times$ faster than the C++ implementation.

CLI standard that describes the runtime, which allows for atomic writes of up the size of native integer, which in case of a program compiled for x64 runtime, while using x64 processor is 64 bits [42], while CUDA offers no such feature. The atomic operations allow using a parallelized Gauss–Seidel scheme, where new values are used as soon as they are computed. As atomic writing and reading is not default behavior for CUDA, instead of using Gauss–Seidel, a Jacobi scheme was utilized in the CUDA implementation. The number of iterations for both Jacobi and Gauss–Seidel method can be seen in Figure 16. The iteration count is iterations per external time step, which is thirty seconds, while internal time steps are five seconds long; therefore almost all of the drops in iterations are six drops deep. It can be seen that the fastest converging results converge in 18 iterations; each internal time step takes three iterations per iteration in the last case. This also indicates that the process is not slowed down by too small internal time steps, as each of the internal time steps still have to do multiple iterations. If number of iterations ever reaches a number where internal time steps are only performing one iteration per time step, increasing the internal time step size could be an option.

The difference between the optimized CPU (.NET) and baseline (.NET) version is explained almost entirely with improved memory access patterns and parallelized
constant processing time. The memory access patterns had a larger impact of those two; the constant processing time was reduced from 100 ms to 25 ms, while the average computation time per time step decreased from above 600 ms to slightly above 300 ms. Direct indexing also increased it slightly, but had a larger effect on the GPU implementation, where direct indexing meant that the index arrays didn’t need to be transferred to GPU memory.

![Figure 16: Iterations per external time step. It can be seen that Jacobi converges slightly slower than Gauss–Seidel. The magnitude of the decrements can be explained by internal time steps.](image)

Additionally, to determine whether the growth in computation time follows the theoretical calculations (i.e. that the increase should be quite linear), a set of experiments were conducted using 0.56 million, 1.11 million, 2.22 million and 4.44 million nodes. The results plotted with computation time as a function of number of nodes for both double and single precision implementation can be seen in Figures 17 and 18 respectively.

To further emphasize the degree of linearity, Figures 19 and 20 show the data as computation time per nodes as a function of nodes. A completely flat line would represent perfectly linear scaling, while a declining line would represent increased efficiency over increased number of nodes, while a rising line means that the implementation gets slower per node, as the number of nodes increases. It can be seen that the double precision version of the CPU (.NET) implementation scales quite well, until 2.22 million nodes, but the jump the 4.44 million nodes actually decreases the efficiency to the level of 1.11 million nodes. The trend, however, is same for all
Figure 17: Average computation time per time step as a function of number of nodes (double precision).

Figure 18: Average computation time per time step as a function of number of nodes (single precision).
three implementations; increasing problem size decreases the average computation time per node, which is in line with Gustafson’s take on Amdahl’s law [18].

The slight increase in computation time per nodes with the C++ double and single precision version are in line with each other. Figure 20, however shows a distinct increase in computation time when doubling the number of nodes from 2.22 million to 4.44 million: the computation time per node decreases, while no such phenomena can be seen in the double counterpart, Figure 19. The results were computed twice, each time using average of four slabs per time step, and the same distinct increase remained. Furthermore, even the initial values using 0.56 million nodes were worse with the single precision version than the double precision version with the .NET-implementation, while the C++ version remained relatively even single/double ratio. Only the GPU version seemed to benefit greatly from computing the results in single precision.

![Figure 19: Average computation time per node as a function of number of nodes. A completely flat line would represent perfect linearity, while a declining line means that the algorithm is more efficient as the number of nodes increases. (double precision)](image)

Additionally linear regression was applied to the data that can be seen in Figures 17 and 18. The slopes that were acquired from that linear regression can be seen in
Figure 20: Average computation time per node as a function of number of nodes. A completely flat line would represent perfect linearity, while a declining line means that the algorithm is more efficient as the number of nodes increases. (single precision)

Table 6. It can be seen that the single precision version of the GPU implementation scales the best with increasing amount of nodes. Only .15 \( \mu s \) is added to the computation time per added node, while the .NET version of code increases by .50 \( \mu s \) per each added node. There seems to be almost no difference between the C++ version while using double or float, but the .NET-implementation actually seems to slow down when computing the values in single precision, especially at larger node numbers.

Table 6: Slopes from Figures 17 and 18. Acquired from linear regression using least squares fit.

<table>
<thead>
<tr>
<th>Type</th>
<th>Single precision [( \mu s / \text{nodes} )]</th>
<th>Double precision [( \mu s / \text{nodes} )]</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>0.1571</td>
<td>0.1925</td>
</tr>
<tr>
<td>CPU (.NET)</td>
<td>0.5088</td>
<td>0.3798</td>
</tr>
<tr>
<td>CPU (c++)</td>
<td>0.3006</td>
<td>0.3095</td>
</tr>
</tbody>
</table>

Considering Figures 19 and 20 as well as Table 6, it can be said the GPU version
scaled the best relative to the computation time per nodes. Both the GPU and the
C++ version benefited greatly from decreasing the floating point precision from
double to single, resulting in decreased computation time and improved scaling with
respect to amount of nodes. The single precision version of the .NET C# version
however scaled worse than its double precision counterpart.

4.1 Parallel efficiency
To estimate the ratio of parallel to sequential code, a simplified set of experiments
were conducted with the .NET C# version of the model. The double version was
utilized, as it had previously shown more stability in the results (see Figures 20
and 19). Models were ran using 0.56 million, 1.11 million, 2.22 million and 4.44
million computation nodes with one, two, three and four cores. The speedup was
computed by running four slabs through 240 time steps, taking an average for the
slabs for each time step. The rationale behind benchmarking per time step as opposed
to benchmarking per iteration is that in the context of this work, time spent per
time step is the single most important statistic for the process control and it makes
calculating speedup for different cases trivial. Hyperthreading was disabled for the
duration of this experiment. The speedup can be seen in Figure 21 and speedup per
core can be seen in Figure 22. The improved scaling at high nodes is explained by
reduced sequential code, reduced constant preprocessing times relative to the problem
size and reduction of other potential overheads relative to the computation time.
Figure 21: Speedup. It can be seen that increased problem size results in increased speedup values over increasing cores.

Figure 22: Speedup per core. The results can be interpreted as efficiency per added core. A horizontal line, where all values are one would mean linear scaling with number of cores and it would require the parallel ratio of the code to be 100% and would allow no overhead.
5 Discussion

The computationally intensive part, iteration between time steps, was parallelized almost entirely, which resulted in nearly 6× speedup compared to the original baseline model. The theoretical performance of the kernels was quite close to the experimental measurements.

Overall the single precision (float) version of the GPU kernel far outperforms CPU implementations and scales better. It is also the furthest away from the theoretical calculations. The double precision version should perform in 50 ms in an ideal environment, while the single precision version should perform in less than 5 ms in an ideal environment. There would seem to be one exception to the float versus double performance; the C# .NET implementation seems to perform worse with floats than with doubles. One possible explanation is that there are some implicit conversions from float to double in the code, for example in the parts to which the author of this thesis did not have access to.

The scaling of all of the versions is relatively linear, as the theoretical calculations would suggest. Essentially each kernel call does a swipe of the node matrix and performs \( N \) operations per each node; therefore the computation time of the model is theoretically \( O(N) \), and quite close to that in practise as well. The GPU version scales the best with increasing number of nodes, while the .NET/C# version scaled the worst. The increased efficiency of the GPU version is at least partially explained by the improved data transfer/kernel computation time ratio: increasing size of input increased allowed each GPU kernel to spend more time computing relative to the time taken to transfer the data to global GPU memory.

It would seem that there is a great deal of constant times: for example the GPU constant time for the single precision version is \( 15 + 30 \) ms alone, whereas the computation time for the whole time step is \( 90 \) ms at the very last iterations: the constant processing times are taking 50 % of the time at the last iterations. One critical notion to make about this work is that as there is virtually no interest to increase problem size as the desired accuracy is enough, Amdahl’s law applies entirely within the scope of this work: even if sequential portion of the code remains constant while parallel size increases, it would not make a practical difference as the required accuracy is already obtained at the current level of problem size (amount of nodes). In addition, one single kernel run is very short, which could cause kernel overhead to have quite high impact. One possibility to reduce the overhead would be to produce the same kernel purely in CUDA C, compile whole project with NVCC [20]. However, the benchmarks that have been done for managed CUDA code seem to indicate that the discrepancy between native CUDA and managed CUDA is quite insignificant [45]. It should be noted that ALEAs publication is for a commercial .NET-product that utilizes wrappers to run native CUDA code while this thesis used the open source project managedCUDA for the same purpose [46]. Empty and minimal kernels were also ran both in C++ (CLI) and using managedCUDA to check
for any potential overhead on kernel launches, but none was noted on this system.

The possibility that there is some overhead in using managed code, however, cannot
be completely overlooked or assumed to be nonexistent when looking at such a large
project with multiple kernels and moving parts, each of which cause a bottleneck
in the solution. The possibility of the overhead of managed code applies both for
the CUDA code used in the .NET implementation and for the C++ code used in
the .NET/CLI implementation. The C++ overhead is present potentially once per
time step as the C# .NET code makes a call that invokes CLI/C++ code, while
the CUDA overhead is potentially once per iteration; considerably worse. However,
the benchmarks show that the kernel can compute e.g. 36 iterations in 70 ms. The
theoretical computation time for that given case (36 iterations) is 50 ms; even if all of
the 20 ms that are within that margin were to be spent on the overhead of launching
the CUDA kernel, the implementation is relatively close to its theoretical maximum.
Furthermore, nvprof shows that achieved FLOPS within a single kernel run are lower
than the theoretical peak throughput, which indicates that this potential overhead
of launching the kernels from .NET environment is very low. However, without
making a complete native version with for example C++ and CUDA C, the potential
overhead of this .NET based solution cannot be properly assessed.

The degree of parallelism is relatively low according to the simplistic benchmarks,
judging from Figure 21, where using 4 cores yielded a 2.17× speedup. Using Amdahl's
Law, the minimum degree of parallelism using the speedup value of 2.17 is 72 %: it is
clear that e.g. speedup of 20× would be extremely far fetched, as per Equation (22)
the parallel ratio should be at least 95 %. Therefore the results achieved in this
thesis, with these specific algorithms being parallelized, the results are within the
expected range. It should be noted, however, that the estimation used for the degree
of parallelism is theoretical and the values should be treated as minimum possible
degree of parallelism in the algorithm: there are multiple possible points of overhead
that can affect the parallelized code not to parallelize entirely. It is clear that the
real ratio of parallel code is much higher, as the GPU version shows speedups up to
6×. However, considering the relatively low value of speedup observed on the CPU
implementation, improvements of magnitudes could not be expected.

As established earlier, the constant processing times are causing a large bottleneck to
the system. The kernel initialization time is quite significant drawback in the GPU
implementation. It could be further reduced if boundary condition initialization was
also done on GPU, therefore eliminating the need of transferring the arrays back and
forth between CPU and GPU between each time step. This could not be done within
the framework of this thesis, as the boundary condition initialization happens on
the slab class, which the thesis writer did not have direct access onto. Furthermore,
the practical limitation of this approach is that as this model is intended for online
process control usage in industrial purposes, the temperature for each time step
needs to be known as soon as it is computed, and for that purpose at least the
temperature vector needs to be copied from GPU memory after each time step. This
could, however, be done asynchronously with the next kernel computation as the data on the CPU side is not required for the calculation itself: if the boundary condition utilized GPU, the kernel would always have access to latest temperature data, as it is computed and stored in global GPU memory. The data from CPU to GPU could also be copied asynchronously, but it would require re-writing the kernel in a way that allows it to only use the data already transferred as efficiently as possible. Potential issue with this method would be increased GPU memory usage, as all of the vectors associated with their respective slabs would have to be kept in the GPU memory throughout the entire process, while with the current version it is possible to keep as many as 30 or as few of the slabs in memory concurrently as needed.

This model makes no guarantees that each time step will be computed within a bounded time. It does have a fall back mechanism, where if the GPU kernel fails, the computation is resumed on the CPU. If more reliable system is desired in the future, one possibility would be to run the model on a more controlled system, for example a very stripped down operating system that only contains what is needed to run the model. Currently it is possible that the operating system decides to use the resources as it best sees fit, which could potentially reduce the computation time. Ideal system would have no services or running processes that could reduce the amount of available computing resources from the model.
6 Summary and conclusions

A speedup of up to $6 \times$ was observed when comparing to the baseline implementation with the GPU implementation, while the CPU implementation resulted in $2 \times$ in the .NET-framework and $3 \times$ improvement with the C++ version. Considering the high constant processing times that could not be reduced within the framework of this work, the results are within the expected range and overall the results are very promising. The constant times include data transfer to GPU memory, computation of boundary conditions and other miscellaneous overhead that include everything else from memory allocations to function call overhead.

The GPU version clearly scaled the best out of all of the implementations, especially when computing with single precision floating-point numbers. Where the C++ and GPU version showed decreased computation time with single precision floating-numbers compared to double precision, the C# version both computed the results slower with the single precision version and scaled worse with increasing number of nodes when using single precision. Therefore the implementation presented in this thesis should prove to be quite useful in the future if large slabs are utilized, or mesh density needs to be increased for more precise results. Interesting observation was that both the theoretical calculations of the computation time and the experimental results show that the computation time scales relatively linearly with increasing nodes.

6.1 Future work

The results seem to indicate that the .NET-environment and specifically the .NET/C# version is not an optimal choice in terms of the performance compared to the C++ version. One possible approach to this issue would be to eliminate the potentially detrimental interfaces, where either CUDA or the C++ is called from .NET as managed code by implementing the whole model in either C++ and adding CUDA kernels to them, or by implementing everything in C++/CUDA C. This could especially reduce data transfer overheads and also result in more simplified overall architecture of the model, where there are less unknown overhead factors.

Further improvements would include computing the boundary condition values on the GPU as well, saving the results to GPU memory and then keeping them there for the whole period of computation. This would have two advantages; very likely increased computation time of the boundary conditions and reduced memory transfer overheads, as the memory would already reside in the global GPU memory when the iterative computation is started. Potential speedup on the base model ($10 \times 0.2 \times 2 \text{ m}$) would be up to 20 % with that method, as it would eliminate the memory transfers completely, that are currently taking 15–25 ms, while the whole computation time is 80–150 ms. Additionally, the whole project could be implemented in an programming environment that compiles native binary code, as it would seem that C++ produces up to $2 \times$ faster results than the just-in-time com-
piled .NET-solution, even though it is invoked within the .NET-based C# framework. Additionally, the literature seems to suggest that explicit versions of parallelized FDM models seem to outperform the implicit versions. One possible line of progress for future would be to implement the explicit version of this model with very small time steps. This version would have the added advantage of giving the process information more frequently, as the explicit version would be forced to use a very small time step in order for it to be stable. The external time step is fixed relative to the furnace, but it could be a worthwhile to benchmark and test different values of internal time steps.

The version of the model presented in this thesis does not guarantee that the system will always converge to a solution within bounded time, it merely suggests that it is very likely that it will always converge to a solution within a certain time. If a very high reliability system is desired, a real-time bounded version of the model could be developed in the future. It is very likely, however, that such system would be significantly slower than the current implementation and much harder to implement and would likely include hardware specific solutions. A possible solution would be to run the model in a very controlled environment (e.g. a stripped down operating system that includes only what is essential to run the model), to reduce any external factors from influencing the computation time.

To further validate the model, a real furnace and process should be utilized alongside with the model. This would allow to fine tune the accuracy of the model so that it is sufficient for industrial purposes and that it does no compute unnecessarily precise values, when the computation time could be utilized for either computing more slabs or for running some of the other models used in the process.

In addition to the model presented in this thesis, additional models could be built to further improve the process. Currently some of the slabs are let to cool down in order to assess their quality: there is potential for a model that predicts the quality of the steel slab during continuous casting: such model could remove or reduce the need to perform quality checks after continuous casting, which would allow the slabs to enter the reheating furnace while they are still much closer to the desired temperature. Additionally, with this model the heating times in the reheating furnace could be reduced in the actual process, as the slab is ready as soon as the temperature hits the desired target; there is no need to keep the slab in the furnace after that. These changes would likely require changes to the whole logistics system, but could potentially significantly decrease energy consumption and streamline the process.
References


A Appendix

Kernel for boundary points:

```c
// load libraries and initialize constants

__device__
double getPsiValue(params) {
    return Equation 34(\(T_i,m\));
}

__device__
double getSigma(int neighborIndex) {
    return Equation 32(neighborIndex);
}

__global__
void calcIterCore(params) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < endBound) {
        // initialize variables
        n = core_index[idx];

        // 1D-3D indexes
        int i = n % nx;
        int j = (n / nx) % ny;
        int k = n / (nx * ny);

        // get sigma values from all six nodes
        // surrounding the current node
        // -z
        if (k != 0) {
            sigma += getSigma(n - nx*ny);
        }
        // -y
        if (j != 0) {
            sigma += getSigma(n - nx);
        }
        // -x
        if (i != 0) {
            sigma += getSigma(n - 1);
        }
        // +x
        if (nx - 1 != i) {
```
sigma += getSigma(n + 1);
}
// +y
if (ny - 1 != j) {
    sigma += getSigma(n + nx);
}
// +z
if (nz - 1 != k) {
    sigma += getSigma(n + nx*ny);
}
while (getPsiValue(T_ind[n] - 1) > sigma) {
    T_ind[n]--;
}
while (getPsiValue(T_ind[n]) < sigma) {
    T_ind[n]++;
}

double gamma = Equations 31 and 34
double beta = Equations 31 and 34

// jacobii scheme: read results from old array
// and then write to a new array
// T1 will be given to kernel as T
// on the next iteration as a starting point
T1[n] = Equation 35(sigma, gamma, beta, T_ind[n]);
diff = T[n] / T1[n];
if (diff < eps1 || diff > eps2)
    isConverged = false;
}