signature supervisor

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DIPLOMA THESIS

GPU Accelerated FVM-DEM Simulation of Laser Based Manufacturing in OpenFOAM®

executed to obtain the academic degree
Master of Science (MSc.) under the supervision of

Univ.Prof. Dipl.-Phys. Dr.-Ing. Andreas Otto

at the
Institute of Production Engineering and Photonic Technologies
Getreidemarkt 9, BA,
9th floor

submitted to TU Wien
Faculty of Mechanical and Industrial Engineering

by

Lukas Bo Heisler, BSc.

12. April 2023

signature student

Acknowledgements

I wish to express my sincerest gratitude to my supervisor Professor Dr. Andreas Otto. His way of working in lively exchange with his PhD and Master students helped me through constant discussion and review of the progress.

Furthermore, I would like to thank my coworkers at the Institute of Production Engineering and Photonic Technologies. I felt very comfortable in the working environment, being able to ask for assistance without hesitation. And special thanks to my co-advisor Michele Buttazzoni for always taking the time to help me with problems or understanding OpenFOAM®.

Last but not least a big thanks to my friends and family. You kept me motivated throughout this time and enriched my life in Vienna. To my girlfriend Luisa and my roommates Bastian and Linus. Thanks for everything. To my parents and brother for their support in all aspects of life.

Declaration in Lieu of Oath

I declare in lieu of oath, that I wrote this thesis and performed the associated research myself, using only literature cited in this volume. If text passages from sources are used literally, they are marked as such.

I confirm that this work is original and has not been submitted elsewhere for any examination, nor is it currently under consideration for a thesis elsewhere.

I acknowledge that the submitted work will be checked electronically-technically using suitable and state-of-the-art means (plagiarism detection software). On the one hand, this ensures that the submitted work was prepared according to the high-quality standards within the applicable rules to ensure good scientific practice 'Code of Conduct' at the TU Wien. On the other hand, a comparison with other student theses avoids violations of my personal copyright.

Lukas Bo Heisler, BSc.

Vienna, 12. April 2023

Abstract

The simulation of laser based manufacturing processes plays a key role in the application in production as well as their improvement. With Computational Fluid Dynamics (CFD), the conservation laws governing laser simulations can be predicted quantitatively. As there are multiple physical phenomena involved, the accurate simulation of processes is computationally demanding. We hypothesized that computations based on the Discrete Element Method (DEM) like Lagrangian particles can be accelerated, using the parallel processing power of graphics processing units (GPU).

In this thesis, an existing numerical solver using the open-source CFD software OpenFOAM® is adapted. All computations regarding the DEM can be outsourced to the GPU, resulting in a reduction of the execution time. The software is validated against the original solver showing quantitatively similar simulation results. Depending on the accuracy of the simulation, the execution time of the GPU-accelerated solver can be more than 100 times faster.

The implementation using discrete Lagrangian particles to simulate the propagation of the laser offers further advantages for the simulation of multiple reflections. Although it is possible to calculate beam propagation and reflections by solving a differential form of the radiative transport equation coupled with ray tracing, the approach gets computationally extremely expensive. With Lagrangian tracers, not only the beam propagation and energy transfer to the mesh can be simulated. Furthermore, it is possible to take into account polarization and wave effects like interference.

Kurzfassung

Die Simulation von laserbasierten Fertigungsverfahren spielt eine Schlüsselrolle bei deren Anwendung und Verbesserung. Mit Modellen der numerischen Strömungsmechanik (CFD) lassen sich Prozesse wie Laserschweißen quantitativ vorhersagen. Auf Grund der Vielzahl der physikalischen Phänomene ist die akurate Simulation solcher Prozesse sehr rechenintensiv. Wir haben die Hypothese aufgestellt, dass Berechnungen, die auf der Diskrete-Elemente-Methode (DEM) beruhen, durch den Einsatz von Grafikprozessoren (GPU) beschleunigt werden können.

Im Rahmen dieser Masterarbeit wird ein bestehender CFD-Solver in der Simulationssoftware OpenFOAM® angepasst. Alle Berechnungen der DEM können auf der Grafikkarte vorgenommen werden, wodurch die Simulationszeit reduziert wird. Die Validierung erfolgt gegenüber dem ursprünglichen Solver und zeigt quantitativ gleiche Ergebnisse. Abhängig von der eingestellten Präzision lässt sich die Simulationszeit um mehr als das 100-fache verkürzen.

Die Ausbreitung des Laserstrahls lässt sich durch Lagrangesche Partikel berechnen. Dieser Ansatz vereinfacht die Simulation von Vielfachreflexionen. Zwar ist es auch möglich, die Strahlausbreitung und Reflexion durch Lösen der differenziellen Strahltransportgleichung in Kombination mit Ray-Tracing zu berechnen. Allerdings ist dieser Ansatz deutlich rechenintensiver. Durch Lagrangesche Tracer kann sowohl die Strahlausbreitung als auch der Energietransfer an das Gitternetz simuliert werden. Weiterhin besteht die Möglichkeit, Polarisation und Interferenz sowie weitere sekundäre Effekte einzubeziehen.

List of Figures

List of Tables

Nomenclature

$\mathbf{I}$	Identity matrix
α_i	Theoretical phase density ratio
$\boldsymbol{\sigma}$	Stress tensor
$\boldsymbol{\tau}$	Shear stress tensor
$\boldsymbol{b}$	Body force per unit mass
$\boldsymbol{S}_d$	Darcy force
$\boldsymbol{S}_g$	Gravitational force
$\boldsymbol{S}_s$	Surface force
$\boldsymbol{u}$	Velocity
Δ	Laplace operator
κ	Thermal conductivity
λ_a	Particle trajectory fraction
λ_c	Particle trajectory fraction from cell centre
$\mathbf{b}$	Source vector
$\mathbf{C_c}$	Cell center
$\mathbf{C_f}$	Face center
$\mathbf{n_f}$	Face unit normal vector
$\mathbf{n}$	Normal vector

$\mathbf{S_f}$	Face area vector
∇	Nabla operator
ν	Kinematic viscosity
ρ	Density
ρ_i	Theoretical phase density
C	Speed of light in vacuum
c_p	Specific heat capacity
Co	Courant number
D	Interpolated laser caustic
E_i	Phase energy
k	Extinction coefficient
l_e	Electron ballistic penetration length
n	Refractive index
p	Pressure
Q_{abs}	Absorbed laser energy
$S_{p,i}$	Thermal pressure coupling
T	Temperature
t	Time

Contents

1 Introduction

In recent years, laser based manufacturing processes have become widely used. Compared to similar production operations, there is generally low throughput times, tool wear and component degradation due to heat. Meanwhile, very high precision and flexibility can be maintained. As with most technologies, the full potential can be utilized when accurate simulation software is at hand. The magnitude of different parameter combinations can not be considered in real-life experiments.

Laser simulations are computationally demanding, as there are multiple physical phenomena interacting with each other, including fluid flow, heat transfer, phase changes and material deformation. With Computational Fluid Dynamics (CFD), the conservation laws governing fluid motion can be predicted quantitatively. The Finite Volume Method (FVM) is a numerical technique, allowing the discretization of partial differential equations like the conservation laws. Key feature is the division of the simulation geometry into a finite number of cells or volumes. Some terms of the conservation equations are transformed into face fluxes. The evaluation of these fluxes is inherently conservative, as the flux leaving a given volume is the same as the flux entering the adjacent control volume. This property makes the FVM the preferred method in CFD [14] [17].

Another description of conservation laws is the Lagrangian approach. Here, fluid-flow and transport phenomena are formulated following a material volume. One application is the Discrete Element Method (DEM). It is generally used for the computation of large numbers of particles in motion. A coupling of both methods has proven to be an effective tool to conduct laser simulations. Yet due to the complexity, simulations running on CPUs can take weeks or months to complete.

The graphics processing unit (GPU) is a highly parallelized computer hardware. It can perform thousands of congruent computations simultaneously. By using the processing power of GPUs, simulation applications can be accelerated, allowing for a faster and more accurate prediction of the behaviour of the system. Such speedups in turn, lead to improved understanding of the processes. This enables better control of the laser-material interaction, higher quality parts and more efficient manufacturing.

In the following thesis, the feasibility of using GPUs in order to accelerate FVM-DEM simulations in OpenFOAM® is investigated.

Project Goals

The aim of this thesis will be the portation of the Lagrangian DEM part of an existing mixed FVM-DEM multiphase solver to be executable on the GPU for performance benefits.
To be more precise, this involves the following tasks:

- Implementation of the 3D particle tracking algorithm in CUDA
- Creation of a static CUDA library which executes all computationally intensive and parallelizable computations for the particle tracking
- Extension of the existing Lagrangian photonic particle, parcel and cloud classes in such a way, that the calculations can be either executed on the GPU or the CPU
- Design of a C++ class which handles all necessary data conversion from the OpenFOAM® framework to the GPU particle tracking algorithm like mesh data, particle position & velocity
- Comply to the OpenFOAM® programming idioms (Object-oriented programming (OOP) and curiously recurring template pattern (CRTP)) as far as possible and develop all software in an easily extendable way
- Testing of the correct implementation by comparing it to a solely CPU based approach in more than one test case
- Benchmark the efficiency, latency and memory footprint, identification of possible bottlenecks
- Documentation of the code through Doxygen in order to be easily expendable for future users

2 Review of Laser Simulation with OpenFOAM®

In the following chapter, the adapted Volume-of-Fluid (VOF) approach in combination with the DEM for solving fluid flow problems with Lagrangian particles is presented. After a brief overview the general model and governing equations are described. Then the solution procedure using the FVM in the C++ toolbox OpenFOAM® is presented.

2.1 Overview

The development of an accurate model for the simulation of laser manufacturing processes is very complex. In order to simulate these processes, a mixture model for the fluid-mechanical multiphase problem is employed, where the Navier-Stokes Equation is solved for a mixture of N phases. This approach is coupled with additional models like beam propagation, laser-material interaction, phase change and surface tension. Fluid simulations are predominantly based on the FVM with the most widely used method being the VOF approach [12]. It enables the observer to fix a volume in space and monitor the change in the properties of the fluid, like velocity, temperature or pressure. A Lagrangian model like the DEM allows the observer to follow a fixed material volume moving in space and time. A mixed FVM-DEM approach hence makes the integration of particles into fluid simulations possible [17]. In the approach described below, particles are used to calculate the laser beam propagation and material interaction. The solver was developed by Otto et al. and fist described in [18].

2.2 Model Description and Governing Equations

As the scope of this work mainly focuses on Lagrangian particle tracking as well as beam-propagation and laser-material interaction, the fluid-mechanical model is

outlined briefly with the former procedures explained in more detail. At the end of this chapter, the overall solution procedure of the utilized multiphase solver is explained.

Continuity and Momentum Equation

The continuity equation describes the transport of the mixture of phases in the volume. In its general form it reads as

$$\frac{\partial \rho}{\partial t} = \nabla \cdot (\rho \boldsymbol{u}). \tag{2.1}$$

The left hand side of the equation denotes the change in time of the density ρ of the mixture. On the right hand side, the divergence of the vector field of velocities $\boldsymbol{u}$ indicates the expansion or contraction of the fluid [9].
The momentum equation relates the forces acting on a volume of fluid to its acceleration

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u}) = \nabla \cdot \boldsymbol{\sigma} + \rho \boldsymbol{b}, \tag{2.2}$$

with $\boldsymbol{b}$ denoting any force per unit mass acting on the body of fluid and $\boldsymbol{\sigma}$ denoting the symmetric stress tensor describing all forces acting within the fluid. The latter can be rewritten in order to emphasize the linear relationship between shear stress $\boldsymbol{\tau}$ and shear rate as

$$\boldsymbol{\sigma} = \boldsymbol{\tau} - p\boldsymbol{I}. \tag{2.3}$$

The body forces can be split up into gravitational and surfaces forces as well as a Darcy source term due to movement restrictions of the solid phases [21]:

$$\rho \boldsymbol{b} = \boldsymbol{S}_g + \boldsymbol{S}_s + \boldsymbol{S}_d \tag{2.4}$$

Energy Equation

In the simulation of multiphase mixtures, all phases inside a cell share a common temperature T which is relevant for heat conduction. Since the convective transport is associated with the individual phase and motion, it is decoupled from the conductive transport and calculated first. Due to the temperature-dependency of the materials heat conductivity and capacity, convection is then calculated by solving the equation

$$\frac{\partial E_i}{\partial t} + \nabla \cdot (\boldsymbol{u} E_i) + S_{p,i} = Q_{abs,i}, \tag{2.5}$$

N times, once for every phase [2]. The total energy is obtained through summation of the individual phase energies E_i. The source term $S_{p,i}$ represents thermal-pressure coupling and $Q_{abs,i}$ denotes the laser energy absorbed by each phase i. Both source terms are distributed energy-conservative among the phases.

Conduction can be calculated after determination of the temperature distribution T_{conv} by solving

$$\frac{\partial \left(\rho c_p T_{conv}\right)}{\partial t} = \nabla \cdot (\kappa \nabla T)\,, \tag{2.6}$$

where κ, ρ, and c_p denote mixture thermal conductivity, density and specific heat capacity, respectively. T_{conv} is the converged temperature value after the convective heat transport.

Laser Beam Propagation and Interaction

As stated above, the propagation of the laser beam is simulated by distributing the laser energy onto 'photon' particles. It must be noted, that not every photon of the laser beam is explicitly simulated, as this would exceed the processing power of even the most advanced high performance computer. The propagation with the speed of light follows a flow field which resembles the caustic of the laser [18]. The caustic describes the intensity distribution (e.g. Gaussian, tophat or Bessel) inside the focused beam.

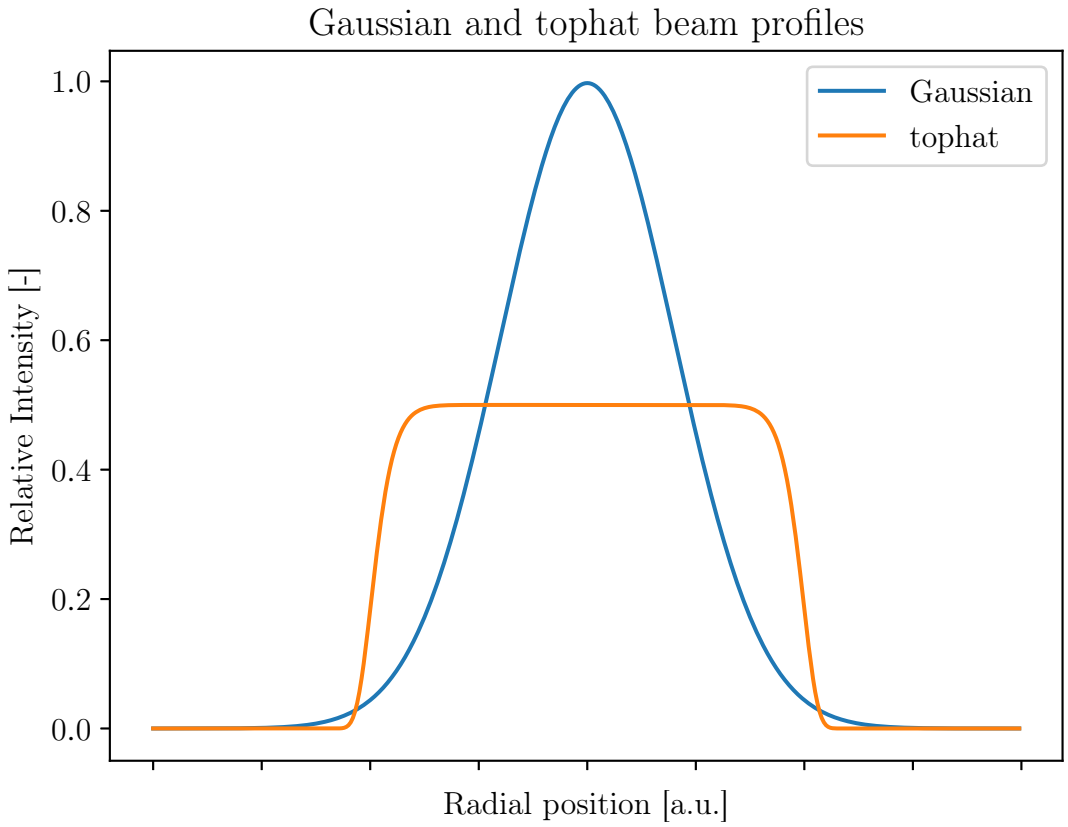

Figure 1: A Gaussian laser beam will have a peak intensity twice as large as a flat top laser beam with the same average optical power.

The particles velocity $\boldsymbol{u}$ is updated each iteration according to

$$\boldsymbol{u} = \frac{C}{n} D \tag{2.7}$$

where C denotes the speed of light in vacuum, n is the refractive index of the current medium and D is the caustic of the laser interpolated to the position of the particle.

A laser beam with the intensity I hitting a flat material surface will be reflected, transmitted and absorbed, such that

$$1 = R + T + A \tag{2.8}$$

defines the energy distribution [13].

The absorption A is calculated using Beer Lamberts law for each medium in the mixture of the cell, weighted by its *phase-normed* α_i coefficient:

$$A = \sum_{i=1}^{N} \alpha_i A_i \tag{2.9}$$

The absorption coefficient consists of a linear part

$$A_{i,lin} = \frac{1}{\frac{\lambda}{4\pi k} + l_e} \tag{2.10}$$

with the extinction coefficient k, wavelength λ and the electron ballistic penetration length l_e. Additional phase specific non-linear absorption involves the multi-photon, free-carrier, cascade and plasma absorption effects. [13]

Reflection at the material interface is calculated based on the incident angle of the particle. For polarized light, there are two reflection coefficients, r_s for perpendicular and r_p for parallel polarization. With θ_1 and θ_2 representing the angles of incidence and refraction, the Fresnel equations read as

$$r_s = \frac{n_1 \cdot cos\theta_1 - n_2 \cdot cos\theta_2}{n_1 \cdot cos\theta_1 + n_2 \cdot cos\theta_2} \tag{2.11}$$

and

$$r_p = \frac{n_2 \cdot cos\theta_1 - n_1 \cdot cos\theta_2}{n_1 \cdot cos\theta_2 + n_2 \cdot cos\theta_1} \tag{2.12}$$

with the refractive indices n_1 and n_2 of the two media. The reflection is composed of the normalized reflection coefficients with the respective proportions of parallel (p) and perpendicular (s) polarized light [13]:

$$R = s \cdot r_s^2 + p \cdot r_p^2 \tag{2.13}$$

For unpolarized light one can assume an equal distribution of the polarization states and the equation simplifies to:

$$R = \frac{r_s^2 + r_p^2}{2} \tag{2.14}$$

Due to the conservation of energy, the transmission T is the portion of the incident power that is not absorbed nor reflected. It arises from equation 2.8. The refracted

particles propagate with a straight linear movement through the new media. The direction is calculated according to Snell's refraction law:

$$\frac{n_1}{n_2} = \frac{sin\theta_2}{sin\theta_1} \tag{2.15}$$

2.3 OpenFOAM® and FVM

In order to simulate flow problems, the derived set of governing equations is combined with a numerical model which can be solved on a computer. The basic idea is to describe continuous physical properties with discrete values. By splitting time into intervals of duration Δt, space into finite volumes and fields into discrete cell values, the partial differential equations that describe physical phenomena can be simplified into sets of linear equations. [9]

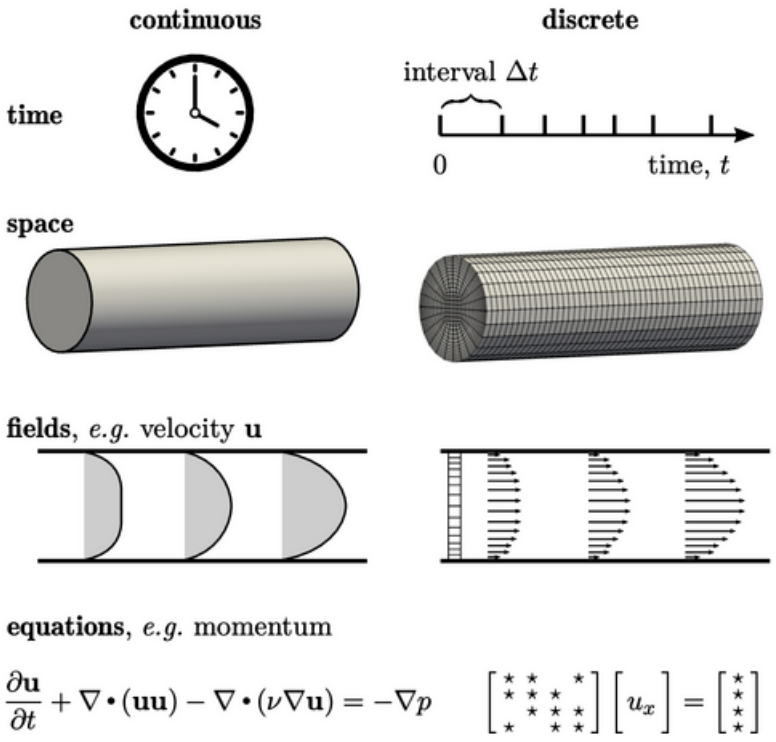

Figure 2: Continuous physical entities are represented by equivalent discrete entities [9]

.

OpenFOAM® is an open-source C++ toolbox for simulating fluid dynamics based on finite volumes. It offers a variety of built-in applications that fall into three categories: pre-processing, solving and post-processing. The pre-processing includes mesh-generation and data-manipulation tools. Solvers are designed to numerically solve a specific problem in fluid (or continuum) mechanics. Post-processing tools like *paraView* allow the solution visualization using a graphical user interface (GUI). Due to the open accessibility of the source code, existing solvers can be improved and adapted easily [11].

OpenFOAM® finds wide applications for fluid flow, heat transfer and multi-physics simulation problems. The Finite Volume Method is used to solve the governing equations of fluid dynamics. FVM is a numerical technique used to solve partial differential equations (PDE). The domain is divided into many small control volumes with a simple geometry. The PDEs are solved by approximating the average values of variables and fields in each control volume.

By utilizing the FVM in OpenFOAM®, we can simulate the multiphysics phenomena that occur during laser based manufacturing processes, and by combining the FVM with DEM, we can model the beam propagation.

3 GPU Computing

To understand the principles of GPU programming, an overview of GPU architecture as well as major differences compared to Central Processing Unit (CPU) architecture is given. Lastly the NVidia programming language CUDA is presented with a context to understand how the coding model works.

3.1 Introduction

In 1965 the engineer and entrepreneur Gordon Moore predicted in an article in *Electronics* that the number of transistors contained on semiconductor chips would roughly double each year. His forecast proved to be true for many years, with the semiconductor industry seeing an increase in component density as well as frequency. In the early 2000's the industry hit what is called the 'power wall'. Cramming more transistors into chips running at ever higher frequencies increased the power consumption exponentially. This increased the CPU's power dissipation beyond the means of inexpensive cooling techniques. There are only two ways of solving this problem: sophisticated cooling techniques and the move to the multi-core CPU design [6].

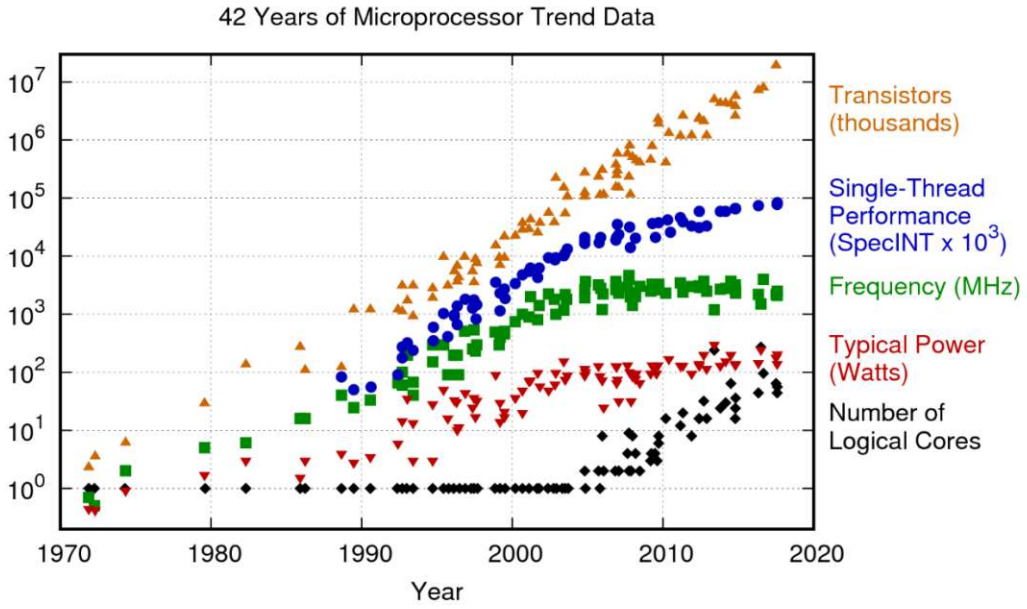

Figure 3: 42 Years of Microprocessor Trend Data [19].

In 2005, both Intel and AMD produced the first multi-core CPUs. As it is visible in figure 3, the number of logical cores has increased ever since [6].

graphics processing units are available since the mid 1980s. Game consoles required dedicated hardware to generate the images in fractions of a second. Since these images are made up of many pixels, GPUs developed into highly parallel compute units with hundreds of physical cores. The image processing methods and hardware began to move into computers and have evolved into the graphical user interfaces everyone is now accustomed to. [5]

By the early 2000s programming languages specifically aimed at harnessing the power of GPUs with OpenGL (Khronos Group) or NVidia releasing Compute Unified Device Architecture (CUDA). CUDA was built as a C/C++ and Fortran extension, allowing the development of GPU-specific code without deepest technical insight. This caused the use of GPUs for mathematical and scientific computing to explode.

3.2 Architecture of Graphics Processing Units

At the highest level, a GPU is a highly parallel computing device, consisting of multiple processing units and a memory hierarchy. The processing units are called

Streaming Multiprocessor (SM) and are connected to an on-chip L2 cache and a high-bandwidth DRAM.

Figure 4: GA10x GPU Architecture [5]

The parallel processing power of a GPU is mostly determined by its Streaming Multiprocessor and cores. The SMs contain several cores which are responsible to execute the instructions. These instructions are executed with the Single

Instruction Multiple Data (SIMD) technique. This allows to process multiple pieces of data with a single instruction concurrently.

Each SM has a small local memory called Shared Memory. It can be accessed faster than the large Video Random Access Memory (VRAM). In order to reduce the load on the VRAM and to hide latency, intermediate results and thread-private variables are stored here.

A thread is a small set of instructions or subset of a process. On the GPU, thread scheduling is implemented in hardware. The scheduler of each SM bundles threads into warps of 32 threads. The threads of a warp are executed simultaneously, they share the same process state and adress space. Because of this, Nvidia refers to this architecture as Single Instruction Multiple Threads (SIMT). In case a warp has to wait for data, its execution can be exchanged by another warp, ready for calculation. Since each warp executes one common instruction at the time, divergent branching in a warp is inefficient. [4][5][1]

3.3 CUDA Programming

The CUDA programming language extends C++ through the definition of GPU-executable kernels. When calling a kernel it is executed N times by N different CUDA threads. As opposed to standard C++, where a function is executed only once by the process. Kernels can be called from the host, which is the CPU process, or from the device, the GPU. In the kernel definition, the specifiers __global__ or __device__ define, where a certain kernel can be called. A __global__ kernel is called using the <<<numBlocks, threadsPerBlock>>> execution configuration. A __device__ kernel is called from inside the device and executed only by the thread that called it. [3]

Let's compare a simple vector addition programmed in C++ and CUDA. The VecAdd kernel is defined globally and called form within the main()-function block. The kernel is invoked with N threads, where N is the number of vector elements to be added. Inside the kernel, each thread sets its thread id, adds the elements from the A and B vector at that index and sets C. Hence each thread performs two *READ*, one *ADD* and one *WRITE* operation.

Listing 3.1: Vector addition CUDA kernel

```
1 // Kernel definition
2 __global__ void VecAdd(float* A, float* B, float* C)
3 {
4     int i = threadIdx.x;
```

```
5       C[i] = A[i] + B[i];
6   }
7
8   int main()
9   {
10      ...
11      // Kernel invocation with N threads
12      VecAdd<<<1, N>>>(A, B, C);
13      ...
14  }
```

In regular C++, the addition is executed inside a for-loop. The index i goes from 0 to N. In each iteration, the elements from A and B are added and set at the index in the C array. It becomes obvious that the work of the function is executed only by one process, as opposed to the kernel call, where the work is split evenly over all participating threads.

Listing 3.2: Vector addition C++

```
1   int main()
2   {
3       ...
4       for(int i = 0; i < N; i++){
5           C[i] = A[i] + B[i];
6       }
7       ...
8   }
```

On first glance, the vector addition using a CUDA kernel seems superior in every aspect. Yet what is not shown in the code snippets is the device memory management. As stated above, host and device have their own separate memory. Kernels operate out of device memory, meaning that the data must first be allocated and copied to the device, before calculations can take place and the result is copied back to the host. The transfer of data between CPU and GPU takes place through a PCIe 4.0 bus interface, offering a bandwidth of up to 64 GB/s. For comparison, data transfer between CPU and DDR5 RAM (dual channel DDR5-6400) can scale up to 102 GB/s and the theoretical memory bandwidth on the GPU is 936 GB/s (NVidia GeForce RTX 3090FE). Moreover, the PCIe bus is a latency source, with a kernel launch latency of $\sim 10\mu s$ as opposed to an on device memory latency of $\sim 100ns$.[5]

In conclusion GPU kernel functions can offer large potential for accelerating computations. The CPU is a low latency component and trumps at serial

processing. Fewer but very versatile cores at higher clock speed are able to execute diverging instructions. The GPU on the other hand is a high throughput component specialized in parallel processing. Running simple instructions without divergent branching on thousands of cores in parallel can make up the slower clock speed and reduced versatility.

4 Particle Tracking on the GPU

In this chapter, OpenFOAM® mesh representation is explained, as its understanding is crucial for the comprehension of the applied particle tracking algorithm. Next, the tracking algorithm itself is presented. It was published by Macpherson et al. ([15]) and implemented in CUDA for GPU accelerated particle tracking in flow fields by Bruenggel ([1]). Since 'photon' particles are not affected by the velocity of the fluid, the provided implementation was adapted. The software now also focuses on the laser-material interaction as a parallel implementation proved very effective. Lastly, the implementation of the algorithm in OpenFOAM® is explained, emphasizing the communication streams and synchronization points between CPU and GPU.

4.1 Mesh Representation

OpenFOAM® offers the mesh-generation application *blockMesh*. It can generate structured and unstructured polyhedral meshes. The mesh is then stored in files of the simulation case directory. The content is explained below. For a more detailed description, refer to [11], [1] and [10].

- **Points:** A list of cell vertex coordinates
- **Faces:** A list of cell faces. The point indices are used to reference the vertices. OpenFOAM® distinguishes between boundary faces and internal faces. Internal faces have an owner and a neighbour cell, whilst boundary faces lie on the boundary of the simulation domain and just have an owner cell assigned
  - **Owner:** Each face is assigned an owner cell. For internal faces, this is the adjacent cell with the lower label. The face normal vector always points in the outside direction of the owner cell
  - **Neighbour:** Internal faces are assigned a neighbour cell. This is the adjacent cell with the higher label. Since not all faces are internal faces, the list of neighbours has less items than the list of owners

- **Cells:** A list of cells. Each cell class is defined by a list of face numbers
- **Boundary:** A list of patches. Each patch can include one or more boundary faces, representing different region of the boundary. This allows for the definition of multiple boundary conditions

This representation of the mesh allows for a static linearization. The essential file contents can be transferred into array containers and stored in the GPUs main memory. Hence the particles can be tracked in global coordinates as well as in their cell occupancy.

4.2 Basic Particle Tracking Algorithm

Consider the situation as depicted in figure 5. A particle is located at the initial position **a** in cell **A** and moves to the end position **b** in cell **B**. During this tracking event, the particle will move through position **p**, where the trajectory crosses face 1 and change into cell **C**. Next it moves along the line **pp'** to position **p'** and change cell again. Finally, the particle arrives at position **b**.

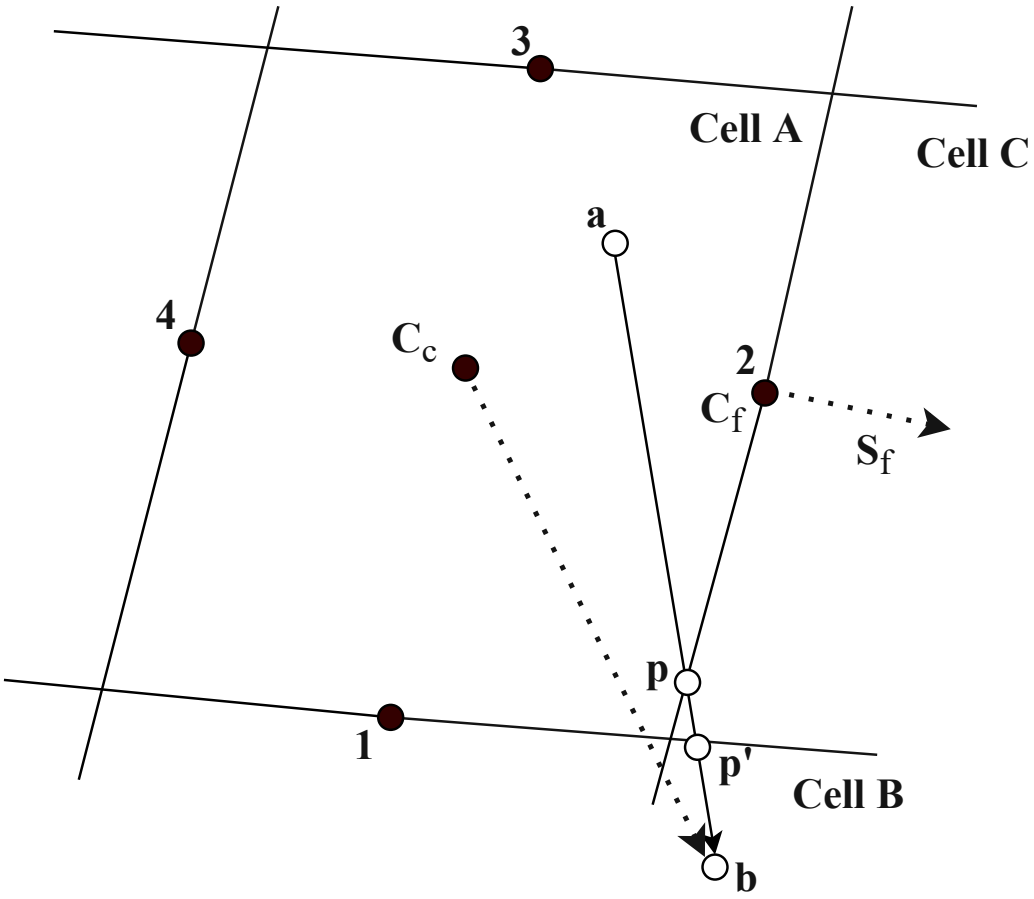

Figure 5: Exemplary particle trajectory with particle moving from position **a** to **b** while crossing two faces

The first part of the motion can be described by the formula

$$\mathbf{p} = \mathbf{a} + \lambda_a \cdot (\mathbf{b} - \mathbf{a}) \tag{4.1}$$

where λ_a denotes the fraction of the path **ab** where the intersection with the face 2 occurs, which is defined by the face centre C_f and the face normal S_f. The position **p** is not known. Since **p** lies on the face

$$(\mathbf{p} - \boldsymbol{C}_f) \cdot \boldsymbol{S}_f = 0 \tag{4.2}$$

substituting Equation 4.1 into eq. 4.2 gives

$$\lambda_a = \frac{(\boldsymbol{C}_f - \mathbf{a}) \cdot \boldsymbol{S}_f}{(\mathbf{b} - \mathbf{a}) \cdot \boldsymbol{S}_f} \tag{4.3}$$

and hence an equation which no longer depends on $\mathbf{p}$. With Equation 4.3 the value λ_a can now be calculated for each face of the cell, that the particle occupies. For each calculation, the respective face centre $\boldsymbol{C}_f$ and the face normal $\boldsymbol{S}_f$ is applied. The face that the particle actually crosses is the face with the lowest λ_a value in the interval $0 \leq \lambda_a \leq 1$. In the example situation from figure 5, face 2 will yield a smaller λ_a than face 1 (or more exact, the plane defined by face 1) since this track fraction is smaller. Once the correct face is found, the particle can be moved onto the face hit using Equation 4.1 and the occupancy information is updated to cell $\mathbf{C}$. The next tracking event is executed in a similar way, with the calculation of the λ_a values for all faces of the cell. Once no face satisfies $\lambda_a \in [0, 1]$, then the position $\mathbf{b}$ must be in the same cell as the particle and it can directly moved to the end point of the motion.

The calculation of λ_a for one particular face can be categorized into three cases. $\lambda_a < 0$ corresponds to the particle moving away from the face. $\lambda_a > 1$ corresponds to the particle moving towards the face but not crossing it and $0 \leq \lambda_a \leq 1$ corresponds to the particle moving towards a face and crossing it.

This basic tracking algorithm might loose track of particles, when the mesh consists of cell faces based on more than three vertices. In that case, not all vertices necessarily lie on one plane. The mesh then stores the face centroid as well as the face surface vector, which in its magnitude represents the *effective* plane of the face. The mesh representation is now no longer space filling and it is possible to loose track of a particle crossing a face close to a vertex [1] [15].

4.3 Modified Tracking Algorithm

The basic particle tracking algorithm can be modified in order to mitigate the problem of loosing track of particles. With reference to figure 5, it is also possible to use other points inside the cell to determine which faces the particle might cross. Starting at any other position inside the cell, a particle would cross the planes of the same faces as if it would have started at position $\mathbf{a}$. It is thus possible to replace $\mathbf{a}$ in Equation 4.3 with $\boldsymbol{C}_c$, which yields

$$\lambda_c = \frac{(\boldsymbol{C}_f - \boldsymbol{C}_c) \cdot \boldsymbol{S}_f}{(\mathbf{b} - \boldsymbol{C}_c) \cdot \boldsymbol{S}_f} \tag{4.4}$$

As in the previous case, if there is no face with $0 \leq \lambda_c \leq 1$ the end point of the motion must be in the same cell. Otherwise, λ_a must be calculated for all faces

where $\lambda_c \in [0, 1]$ and the particle is moved onto the face with the smallest λ_a. With $\lambda_m = min(1, max(0, \lambda_a))$ Equation 4.5 is then used to move the particle to position $\mathbf{p}$.

$$\mathbf{p} = \mathbf{a} + \lambda_m \cdot (\mathbf{b} - \mathbf{a}) \tag{4.5}$$

Below, the complete tracking algorithm can be seen as pseudo code. This was taken from [15] (Algorithm 1).

Algorithm 1 Modified particle tracking algorithm

```
while particle has not reached end position b do
    find set of faces F_i for which 0 ≤ λ_c ≤ 1
    if size of F_i = 0 then
        move particle to end position b
    else
        find face f ∈ F_i for which λ_a is smallest
        move particle with p = a + λ_a · (b − a)
        set particle occupancy to neighbouring cell of f
    end if
end while
```

The algorithm can still cause particles to get stuck. When moving a particle onto a face and calculating all λ_a, the calculation might yield the same face again, causing the particle to move back into the previous cell. Consider the situation as depicted in figure 6.

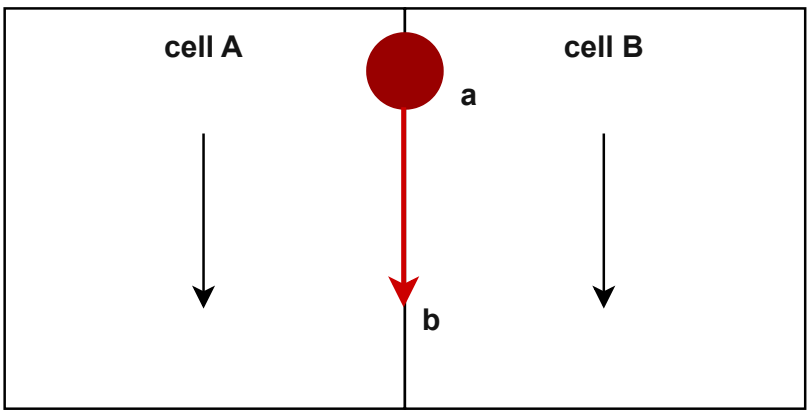

Figure 6: A particle getting stuck on the face between two cells, when the trajectory lies on the *effective* plane of the face

Since the trajectory of the particle lies on the *effective* plane of the face between the cells **A** and **B**, the algorithm will always yield the same face. This will cause the particle to flip from one cell to another with extremely small λ_a values, hence making no progress in its Lagrangian time step. The problem can be avoided by implementing an ϵ-environment or by disabling the face for the next calculation. In a structured mesh, the rotational axis of a laser beam might lie exactly on this plane. An implementation of the particle algorithm with an ϵ-environment helped with no particles getting stuck, but it also caused unphysical behaviour, as some particles remained longer in the simulation domain as the ϵ-environment slowed them down. Thus in the actual implementation, a face can not be crossed twice.

4.4 Implementation in OpenFOAM®

The actual implementation of the algorithm is more complicated. Before, the velocity of the particle was assumed to be constant during all events of the Lagrangian tracking step. In fact this does not describe the physical behaviour in the beam. Instead, as stated in Section 2.2, the propagation follows a flow field resembling the caustic of the laser beam. Consequently, the particles velocity depends on the velocity of the cell it occupies and changes once the particle changes cell (or reaches an interface). The motion of a 'photon' particle during one Eulerian time step describes its track from **a** to **b**. This motion is broken up into several smaller Lagrangian time steps. Figure 7 shows a particle at the beginning of the Eulerian time step at position **a**. The end position is estimated using the laser direction field of the actual cell, marked as $\boldsymbol{b_1}$. Once the cell changes, the end position is re-estimated using the laser direction field from the new cell.

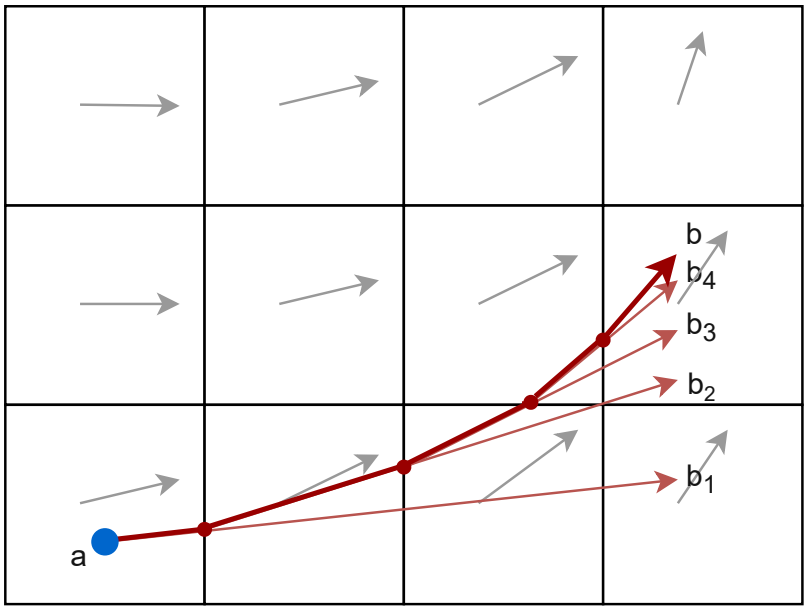

Figure 7: A particle trajectory through multiple cells with the cell field influencing the particles path causing the end position to change after each tracking event [15]

The thick red line shows the actual trajectory of the particle. The light red arrows point to the end position of each respective sub step [1]. A Lagrangian step is limited by two factors. For one, the particle crosses a face and changes occupancy to another cell, making it necessary to reestimate the velocity. Secondly, the Courant–Friedrichs–Lewy condition must be fulfilled:

$$Co = \frac{u\Delta t}{\Delta x} < Co_{max} \tag{4.6}$$

where Co, u, Δt and Δx denote the dimensionless Courant number, the magnitude of the velocity, the time step and the length interval between mesh elements respectively.

5 Software Development

The tracking engine is built upon the `photonParticle` and `photonCloud` library wich is based on the regular OpenFOAM® Lagrangian particle library. The `gpu` switch toggles the execution of the particle tracking and interaction on the CPU or GPU. The CUDA code must be compiled separately using NVidias `nvcc` compiler as shared library and is linked to the OpenFOAM® solver.

Software Design

The communication between the solver and the shared library respectively the GPU takes place using two structs named `StaticHostData` and `TimeDepHostData`. Both contain several `std::vector` container, the former encapsulating all static mesh data and the latter the dynamic mesh and particle data. The static struct is used before the first tracking event and after each mesh refinement. The dynamic struct is altered every iteration, both by the CPU solver and the GPU.

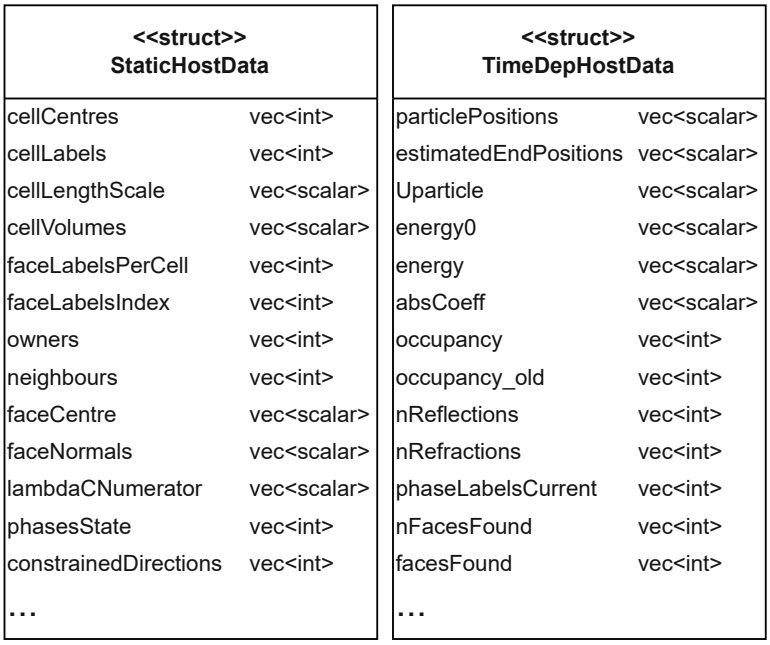

Figure 8: C++ structs for data communication between CPU and GPU

The `std::vector` containers are referenced by a custom wrapper class `cuvector` which encapsulates the container in host memory as well as a pointer to the device memory. Furthermore, basic data transfer functionality like upload and download to device memory is possible via member functions.

Mesh data

All mesh data is stored in a `FlatMesh` class. This class compares to the mesh classes of OpenFOAM® in terms of member variables and basic functionality. Furthermore, it is used to calculate the λ_C-numerator for the modified tracking algorithm, cell neighbour relations as well as encapsulate the transfer of mesh data [1].

OpenFOAM® meshes are stored in linked-list containers, which are not suitable for parallel processing on the GPU. Thus the lists are flattened into fixed size arrays. Vectors are stored using the CUDA `vec3` and `int3` data types, allowing for standard array indexing using the cell or face indices. The mesh data is stored in such a way, that structured and unstructured meshes can be used, e.g. meshes where cells can have varying number of faces. The cell labels are not stored, since

they go from 0 to the number of cells -1. The association of face labels to each cell and vertex labels to each cell was taken from [1] and works the same way for both connectivities:

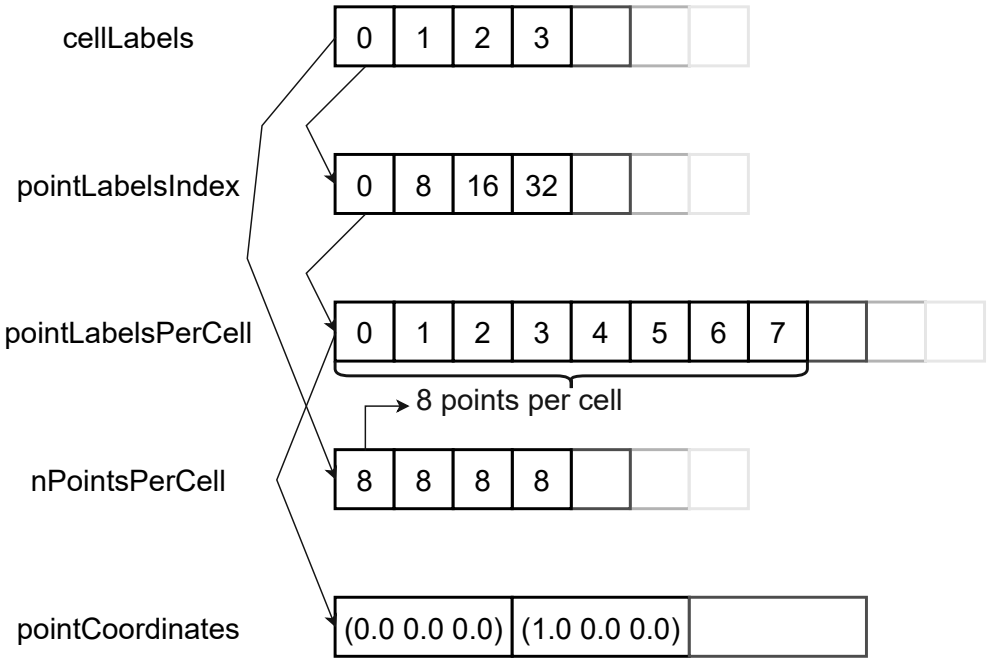

Figure 9: Vertex connectivity arrays allow use of unstructured and refined meshes.

Figure 9 illustrates the connectivity mapping from cell labels to the point labels of the cell. The point labels are stored in the `pointLabelsPerCell` array. This array has more entries than the number of points in the mesh, as each vertex can belong to more than one cell. Using the point labels, the respective coordinates can be obtained from the `pointCoordinates cuvector`.

Particle Engine

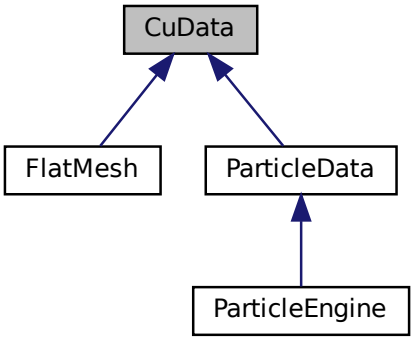

Figure 10: Inheritance diagram for CuData class.

The main body of the software is the `ParticleEngine` class. It inherits from `ParticleData`, a wrapper class similar to the `ParticleEngine` class with all relevant functionality of the GPU particles. The `FlatMesh` and `ParticleData` class inherit from the `CuData` class which handles the conversion from `std::vector` to the CUDA `vec3` and `int3` data types.

Each iteration of the OpenFOAM® solver triggers the execution of the global `runStep` kernel. The basic procedure can be seen in figure 11.

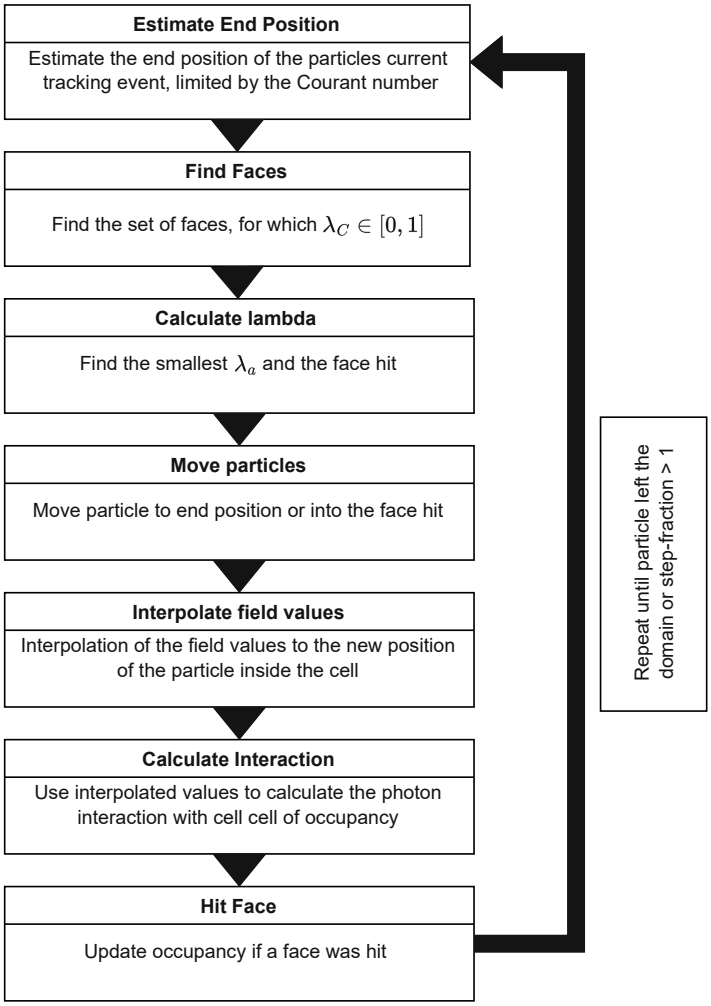

Figure 11: Procedure of the `runStep`-kernel with all device kernel calls.

The execution configuration is calculated such that there is one kernel thread for each particle in the cloud. The big kernel launches device kernels to complete the different substeps of the tracking event and returns after all threads finished the Eulerian time step for their particle and successive child particles.

Field Interpolation

Several scalar and vector valued fields are necessary for the beam propagation and material interaction calculation on the GPU. The values of the fields are

defined only in the cell centre of each cell. In order to interpolate this value to the exact position of a particle inside a cell, the field is interpolated according to the OpenFOAM® `interpolationCellPoint` functionality [8]. At first, the cell centre values are interpolated to the vertices of each cell. The vertex values are influenced by the centre values of all cells that the vertex is part of. Next, the tetrahedron containing the particle is determined. The cell is split into space filling tetrahedron, each consisting of three face vertices and the cell centre. The barycentric coordinates of the particle inside the tetrahedron are determined according to [7]. Using the coordinates as weights and multiplying them with the vertex and centre field values, the field can be interpolated to any position of the particle inside the cell [10] [11].

Dynamic Mesh Refinement

Adaptive or dynamic mesh refinement refers to the adaption of the accuracy of the solution within certain sensitive regions of the simulation domain during the time the solution is calculated. Whereas in a static mesh, the solution is calculated in equidistant or predetermined spacial steps, a dynamic mesh can react to very large gradients by spacing the grid more finely in regions where higher accuracy is needed. The `FlatMesh` class uses the point, face, owner, neighbour and cell lists (see subsection 4.1) and stores the data in fixed length arrays on the GPU. A mesh refinement changes the length of the lists as well as the connectivity of the face and vertex indices and makes a re-initiation of the `FlatMesh` necessary. As there is no variable length `std::vector` equivalent container available on the GPU, the memory space is freed and reallocated every time the mesh changes. This naturally leads to memory transfer overhead. Additionally, the λ_C-numerator is calculated again for all cells.

6 Benchmarks and Comparison

The main goal of this work is to make use of the GPU parallel efficiency in order to speed up the simulation run time. A comparison between the existing CPU based approach and the GPU accelerated solver thus consists of two parts. First, the simulation result should be at least quantitatively similar. Although the CPU solver uses a slightly different barycentric particle tracking algorithm, the particles trajectory should be almost identical. Second, the execution time of the GPU solver should be faster. More specifically, the execution time of the DEM part of each time step will be compared, as the rest of the solver is identical.

In this chapter, four test cases will be evaluated. First of all, the GPU solver is validated against the CPU solver. Then, a simple case with multiple refractions and a static mesh is simulated. Next, the effect of mesh refinement and the successive reallocation and transfer of memory is examined. Finally, a realistic pulsed laser drilling simulation is conducted.

6.1 Validation

In order to visualize diverging results, the CPU and GPU solver are used in a simulation with a 2D meshing of 12000 cells. During laser pulses, 2000 particles are injected into the domain per time step. After 450 time steps, the simulation results are compared.

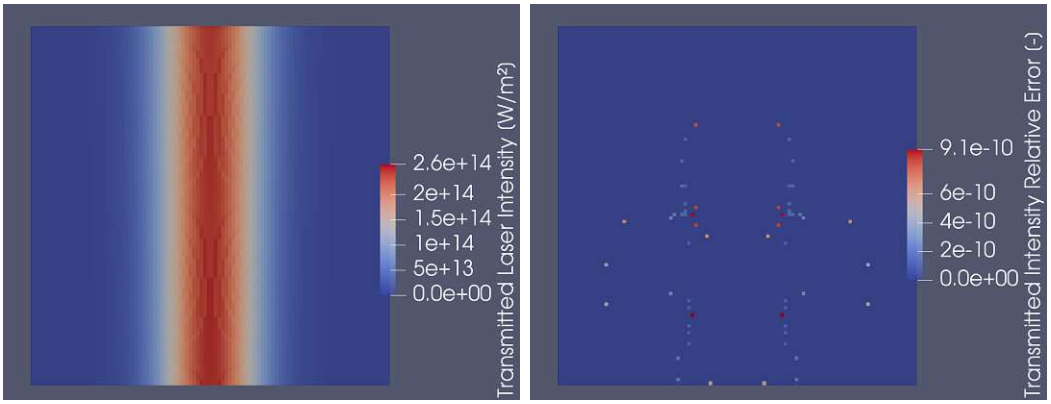

Figure 12: Transmitted intensity field for CPU (left) as well as the relative error between the CPU and GPU field (right)

The transmitted laser intensity in the domain can be seen in figure 12. On the left, the field calculated by the CPU solver shows the intensity of the laser beam. On the right, the relative error per cell between the CPU and GPU solver implementation is visible. Qualitatively, both simulations are the same. The maximum relative error is below 0.0000001% with less than 100 cells affected. It is explainable by particles changing cell at different times in the simulation, due to the different tracking algorithms in use. The error in the intensity field does not accumulate over time, as there are time steps with no particle injections causing the intensity to be zero in all cells.

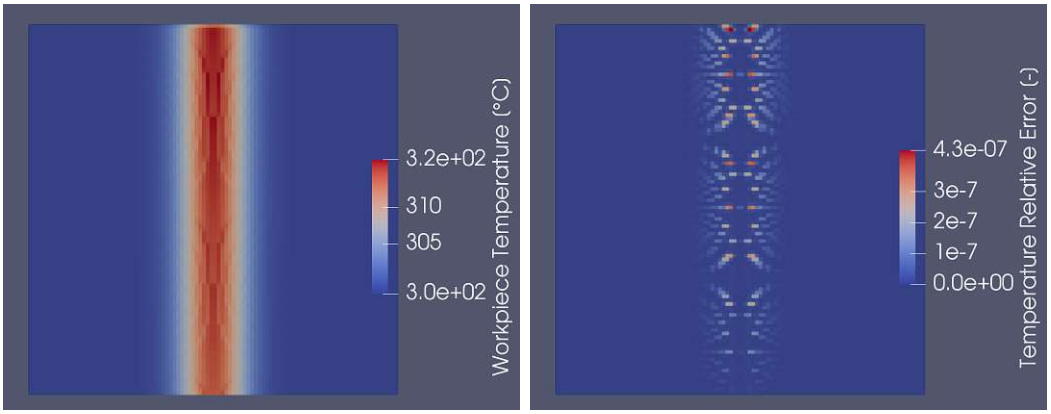

Figure 13: Work piece temperature field for CPU (left) as well as the relative error between the CPU and GPU field (right)

The temperature field as well as the relative error in cell temperature between both solvers can be seen in figure 13. Here, the error does accumulate. After 450 steps, the maximum relative error is less than 0.00005%. It is caused by minor divergences in the trajectories of the particles accumulating over time.

Both results show a negligible divergence between the two solvers. Such small errors will not noticeably affect the simulation.

6.2 Multi-Refraction at Static Mesh

Small Mesh

In this simulation, a 1mm thick plate of borosilicate glass is exposed to a laser beam for 10 nanoseconds. The meshing of the domain is coarse with less than 4000 cells. The number of refractions is set to 2. This results in a maximum number of 2 newly created particles for each initial particle.

On domains composed of relatively few cells, simulating with multiple CPU cores does not make any sense, as the performance gain does not outweigh the communication overhead between threads. Hence on the small domain, the pure single core CPU performance is compared to the GPU performance. With less than 4000 cells, the size of the mesh is only 2.76 MB and only about half that size has to be transferred to the GPU each time step, including e.g. the laser direction cell and point fields. The data size for each particle is around 0.55 kB. Thus the particle data is larger than the mesh data when simulating with more than 2000 particles.

Cells	Mesh data	Particle data (1 particle)	Particle data (100k particles)
4000	2.76 MB	0.55 kB	5.15 MB

Table 1: Data size of mesh and particle data for small mesh benchmark

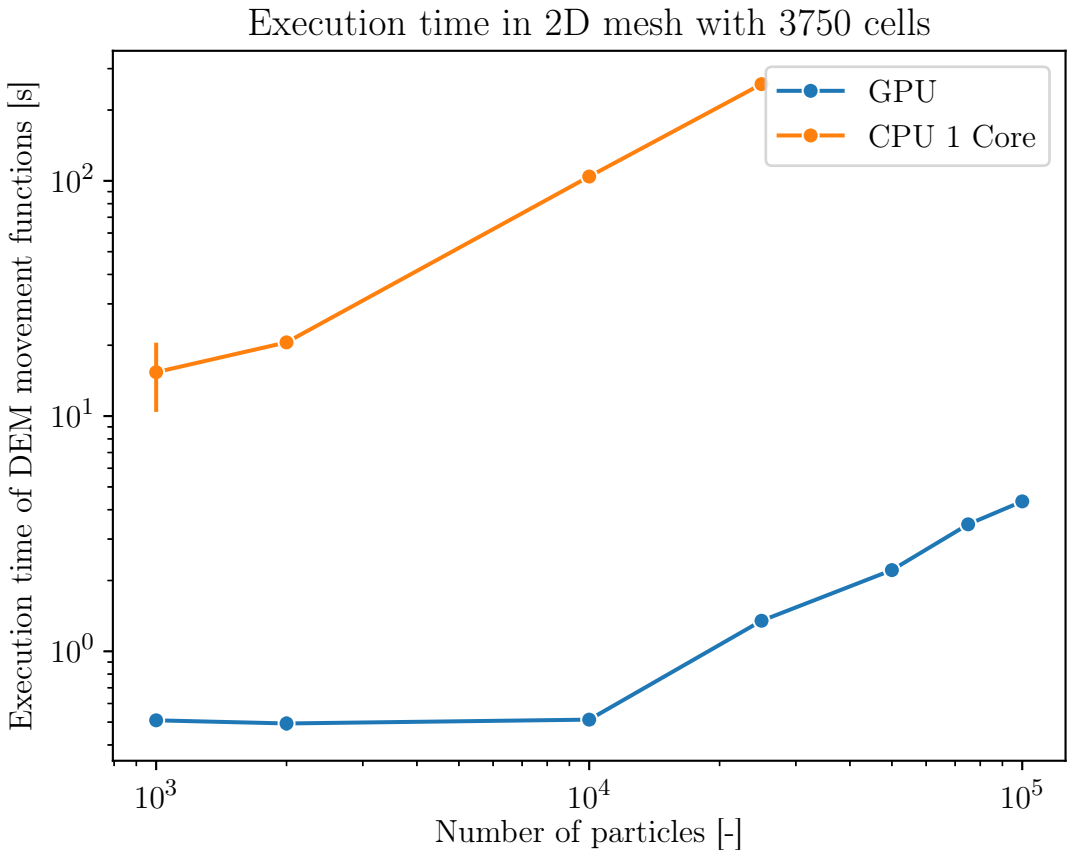

Figure 14: Execution time for particle tracking and interaction on a small static mesh

Figure 14 shows the execution time for the particle tracking and mesh interaction function. The simulation was conducted multiple times, thus the average value as well as the standard deviation is displayed. The performance benefit when using the GPU compared to a single core CPU execution is clearly visible. Already at 1000 particles, the GPU is more than 20 times faster. At 25k particles, the execution time on the CPU is 255 seconds and 1.5 seconds on the GPU. Above around 10k particles, the GPU scales linearly with the number of particles. This can be explained by the maximum number of threads for the given program being reached. This causes single threads to calculate multiple particles and thus linearly increases the workload of the slowest thread.

Large Mesh

Most simulations utilize multiple CPU cores when they can be effectively parallelized, due to reaching a threshold size. The parallelization shares the workload across multiple cores, allowing for faster processing. However, due to dependencies between calculations or iterations, a certain communication overhead increases the run time.

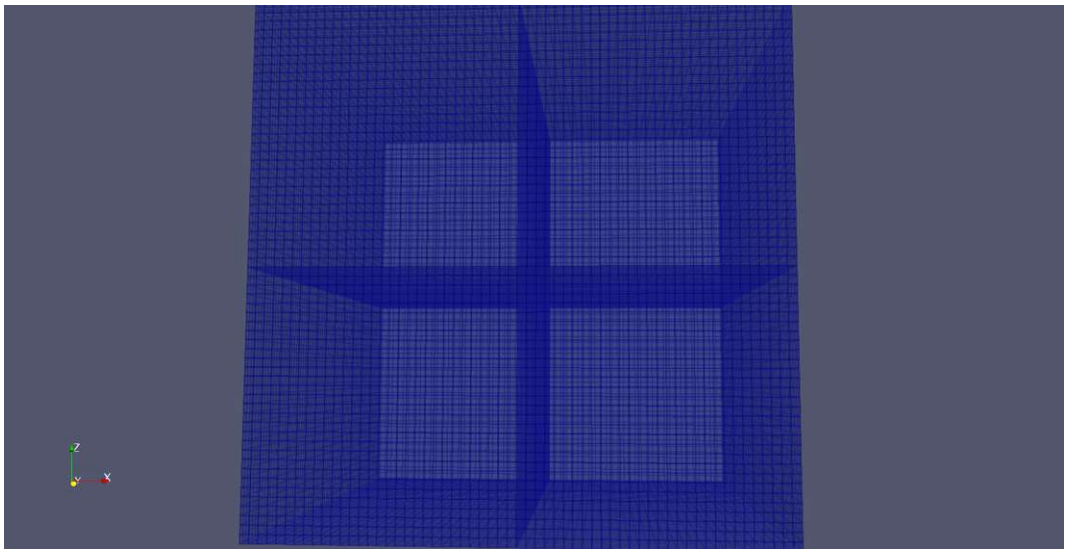

Figure 15: Decomposition for four CPU processes divides the domain through the optical axis of the laser.

Multi-core efficiency of the CPU solver is compared to the GPU solver by decomposing a 3D domain into 2 and 4 subdomains respectively. Figure 15 shows the decomposition along the x- and z-axis, resulting in four equally sized subdomains around the optical axis of the laser. The 3D mesh consists of 252000 cells, thus the data traffic between CPU and GPU is considerably larger, with around 250 MB per iteration for the mesh alone. Furthermore the mesh size is around five times larger compared to the particle data for 100k particles. This leads to this simulation configuration being rather memory bound, at least for smaller amounts of particles. The domain size is 70 µm in each direction.

Cells	Mesh data	Particle data (1 particle)	Particle data (10k particles)
252k	250 MB	0.55 kB	5.15 MB

Table 2: Data size of mesh and particle data for large mesh benchmark

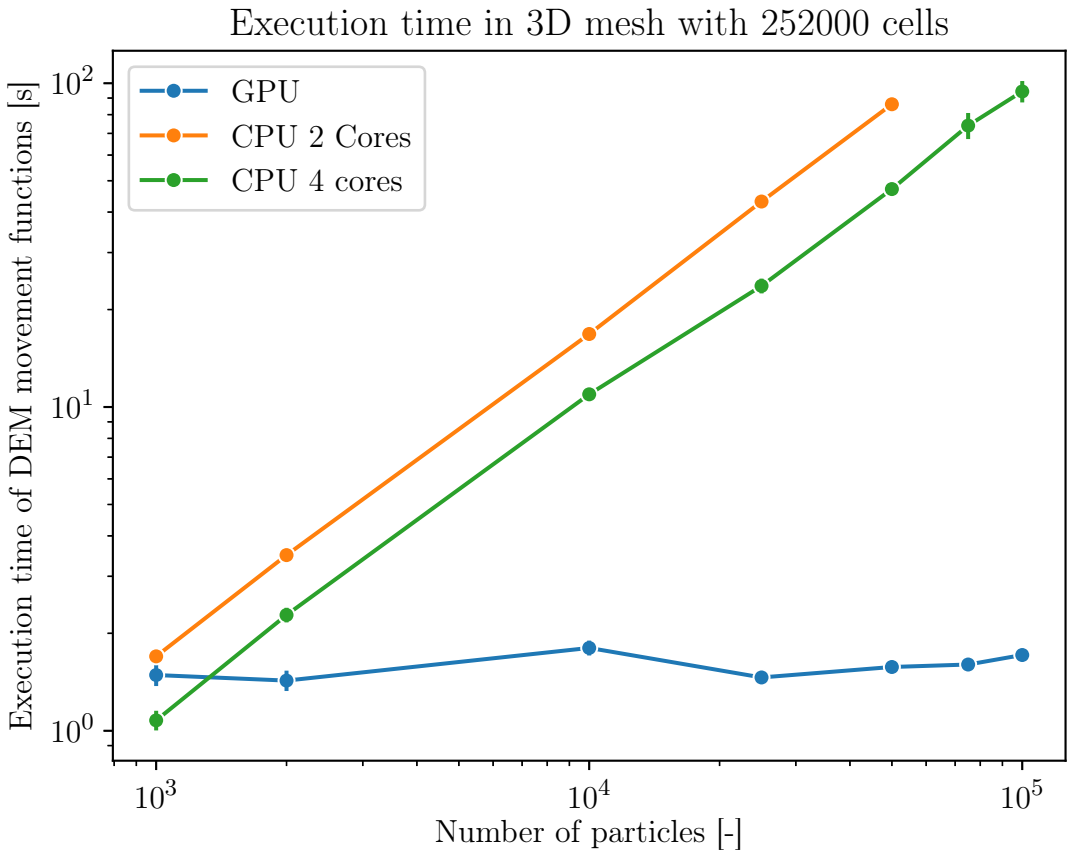

Figure 16: Execution time for particle tracking and interaction on large static mesh with multi-core CPU and GPU.

The execution time plotted over the number of particles can be seen in figure 16. Even for few particles, the DEM part of the code takes more than 1.5 seconds to complete. With an increasing number of particles, the duration stays the same. As expected, the kernel execution is not a bottleneck with a large mesh as the function is limited by the bandwidth between CPU and GPU.

The multi-core simulation time for 2 and 4 CPU cores increases linearly with the number of particles. The domain decomposition is very fair in terms of computational demand per process. Hence the 4-core execution is almost twice as fast as the 2-core execution, showing almost perfect scaling. The communication overhead between processes is evidently very low. Depending on the CPU hardware, this effect can be scaled up. The GPU execution time is still significantly faster.

In conclusion, the execution time for the particle tracking using the GPU is faster than the multi-core CPU when using around 1500 particles or more.

6.3 Laser Exposure on a 3D Domain with Mesh-Refinement

In order to validate the compatibility of the algorithm with mesh refinement and test its influence on the particle tracking algorithm, the 2D test case from section 6.2 is adapted to a 3D mesh. In the unrefined base state, the mesh consists of 16200 cells. Every 5 time steps, the cells of the borosilicate glass work piece are refined, up to three times in total. Hence one base hexahedron is refined into 8, 64 and then 128 smaller cells. The simulations were conducted with just a single particle such that the results are not influenced by large data traffic for the particles.

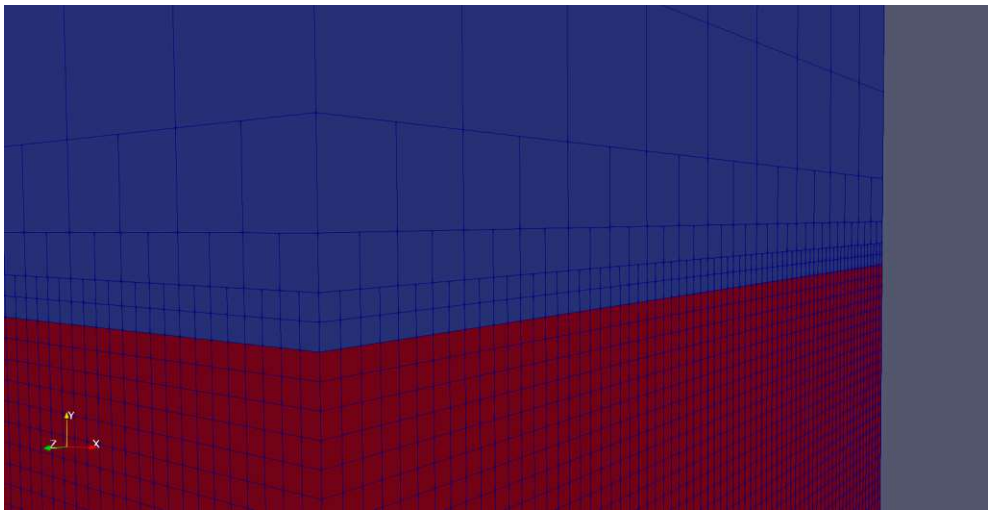

Figure 17: Unrefined surrounding phase (blue) at the top and three times refined borosilicate glass (red) at the bottom.

	Base state	1st Refinement	2nd Refinement	3rd Refinement
Cells	16200	29000	124000	868000
Mesh data	9.64 MB	16.82 MB	70.78 MB	488.63 MB

Table 3: Number of cells and size of GPU mesh depending on number of refinements.

With the number of cells increasing to 29000, 124000 and 868000 after the respective mesh updates, the amount of data to be send to the GPU for each time step increases from around 10MB up to 500MB. Furthermore, the refinement overhead for freeing and reallocating the memory space as well as the recalculation of the λ_c-numerator increases the execution time of the GPU function call, although just in the time steps of the mesh refinement.

The effect of the mesh refinement is clearly visible in figure 18. The time steps 5, 10 and 15 show a spike in execution time compared to the previous time steps. Especially the last mesh update to the highest refinement is computationally expensive with a one-time costly λ_c-recalculation for all cells. After each mesh update, the execution time increases significantly because of the increased number of cells.

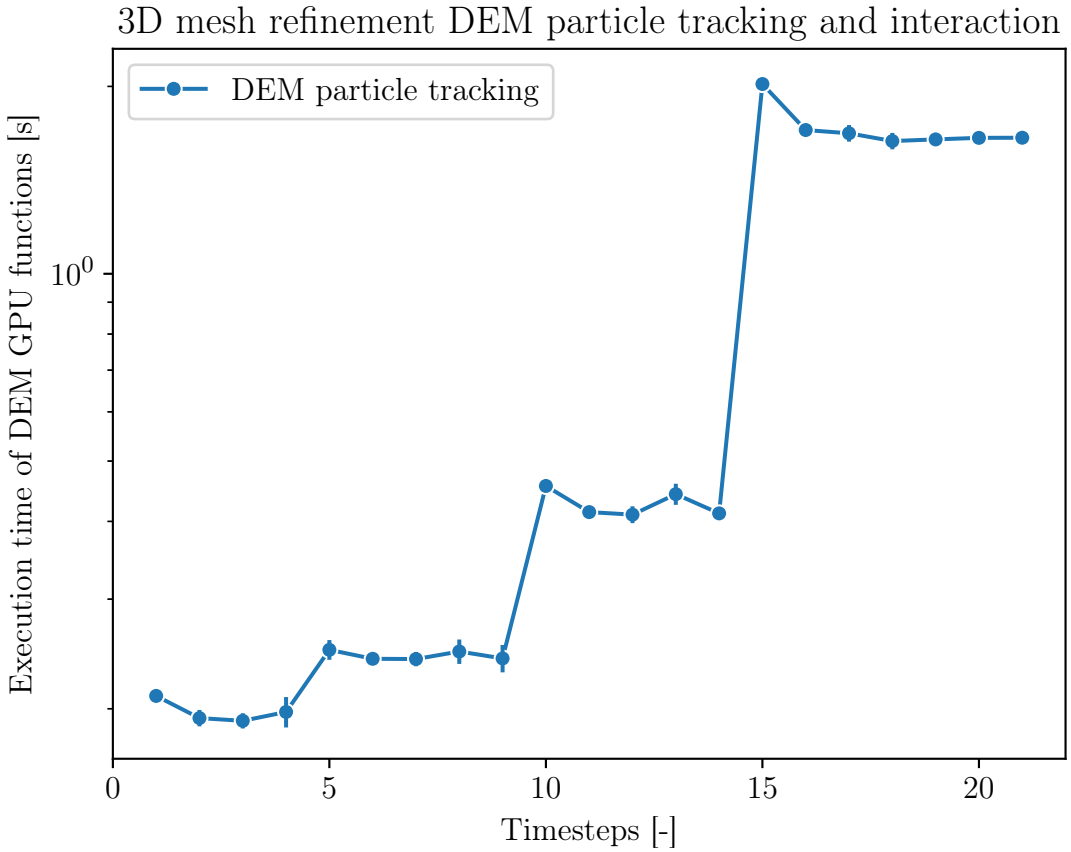

Figure 18: Execution time for `moveGPU`-function in each time step.

In order to evaluate the costs and benefits of the mesh refinement, it is necessary to set it into perspective with the overall execution time of the simulation.

Figure 19 compares the DEM-parts share to the overall execution time for each time step. The size of the mesh does affect the CPU part of the solver as well, not only due to increased data traffic for the GPU. The fraction of the DEM calculations are decreasing with an increasing number of cells. In the time steps where the mesh is refined, the share of the GPU functions execution time to the overall execution time is smaller than usual. Taking these results into consideration, the GPUs contribution is around one order of magnitude smaller than the general cost for a mesh refinement.

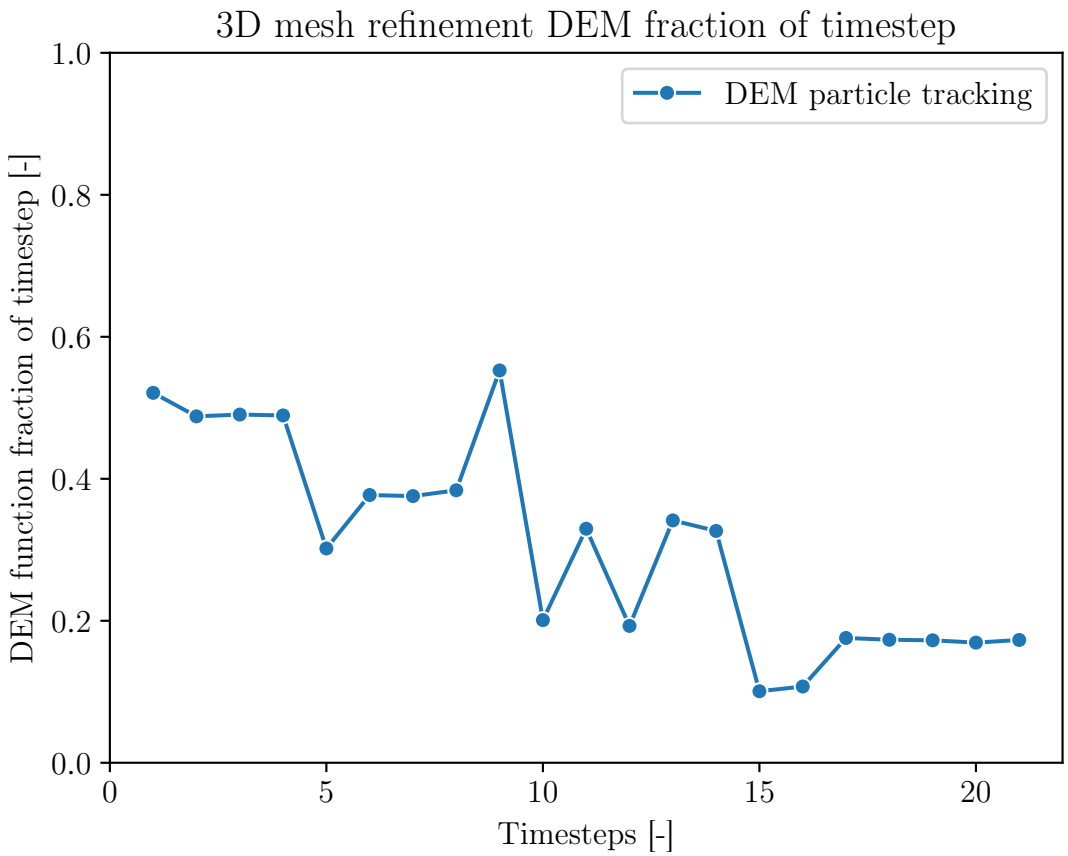

Figure 19: The fraction of the `moveGPU`-function compared to overall time step.

It must be noted, that a mesh refinement does generally not affect a whole domain region like the work piece at once. More likely is a refinement at a small region of interest, like the keyhole of a welding process. Even with a moving keyhole, the overall size of the refined region does not change a lot. Such refinements might not affect the OpenFOAM® mesh as much, due to the linked-list storage containers. The immutable array structure of the GPUs `flatMesh` on the other hand is affected not by the size of the refinement region but the overall mesh size. With the array lengths and connectivity of the face and vertex indices changing, the whole `flatMesh` has to be reinitialized. Yet given the GPUs share to the overall execution time of around 25%, the update overhead does not really affect the run time, especially when the refine interval is not too small.

6.4 Borosilicate Glass Drilling

The final test case is a realistic borosilicate glass drilling simulation using a pulsed ultrashort green laser source. The general simulation parameters are listed below:

Wavelength:	$515\,\mathrm{nm}$
Pulse energy:	$2.333 \times 10^{-6}\,\mathrm{J}$
Pulse duration:	$28 \times 10^{-12}\,\mathrm{s}$
Pulse frequency:	$30 \times 10^{6}\,\mathrm{s}^{-1}$
Pulse shape:	Gauss
Beam shape:	Gauss
Spot size:	$12 \times 10^{-6}\,\mathrm{m}$
Focus height	$-260\,\mu\mathrm{m}$
Work piece material	BK7 borososilicate glass
Work piece dimensions (x,y,z)	$(0.25, 0.5, 0.25)\ \mu\mathrm{m}$

Table 4: Simulation parameters for borosilicate glass drilling case

Even with a coarse simulation domain of around 40k cells, the number of particles necessary to achieve adequate results without undersampling the propagation of the laser is around 100k particles. The results from Section 6.2 show that it is not feasible to compute such complex simulations on the CPU, using the DEM approach. Thus symmetric boundary condition is applied assuming that the model is axisymmetric with respect to the optical axis of the laser. It is hence possible to simulate a quarter of the domain under the assumption that physical variables of the flow behave symmetrical. The laser beam and simulation domain can be seen in figure 20.

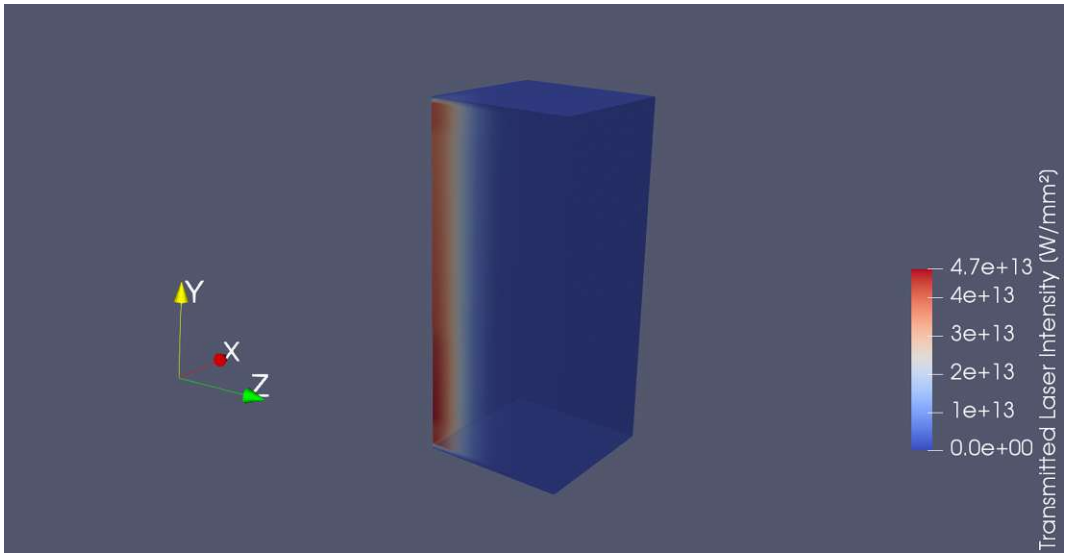

Figure 20: 3D view of pulsed borosilicate drilling simulation. The transmitted laser intensity shows the propagation of the beam though the domain.

The focus height of the laser is clearly below the work piece. The idea is to induce multi-photon absorption at the bottom surface of the glass. The absorption will then increase from bottom to top, creating a column of molten glass. Due to the opacity of the molten glass, material vaporization will begin at the top surface. The vaporization creates a recoil pressure, ejecting the molten glass from the bottom of the work piece [20].

The case is simulated three times. Two simulations are conducted without refraction. The transmitted particles follow the caustic of the laser all the way through the domain. One version does consider an internal reflection at the bottom glass surface , the other one does not calculate internal reflections. The third simulation does consider refraction with straight-linear movement according to Snells refraction law (equation 2.15) as well as one internal reflection. The features are listed in table 5.

Simulation	A	B	C
External reflection at top surface	✓	✓	✓
Internal reflection at bottom surface	✗	✓	✓
Refraction	✗	✗	✓
Color code	blue	green	red

Table 5: Simulation features for borosilicate glass drilling case

Figure 21 shows the temperature of the work piece along a line parallel to the optical axis of the laser.

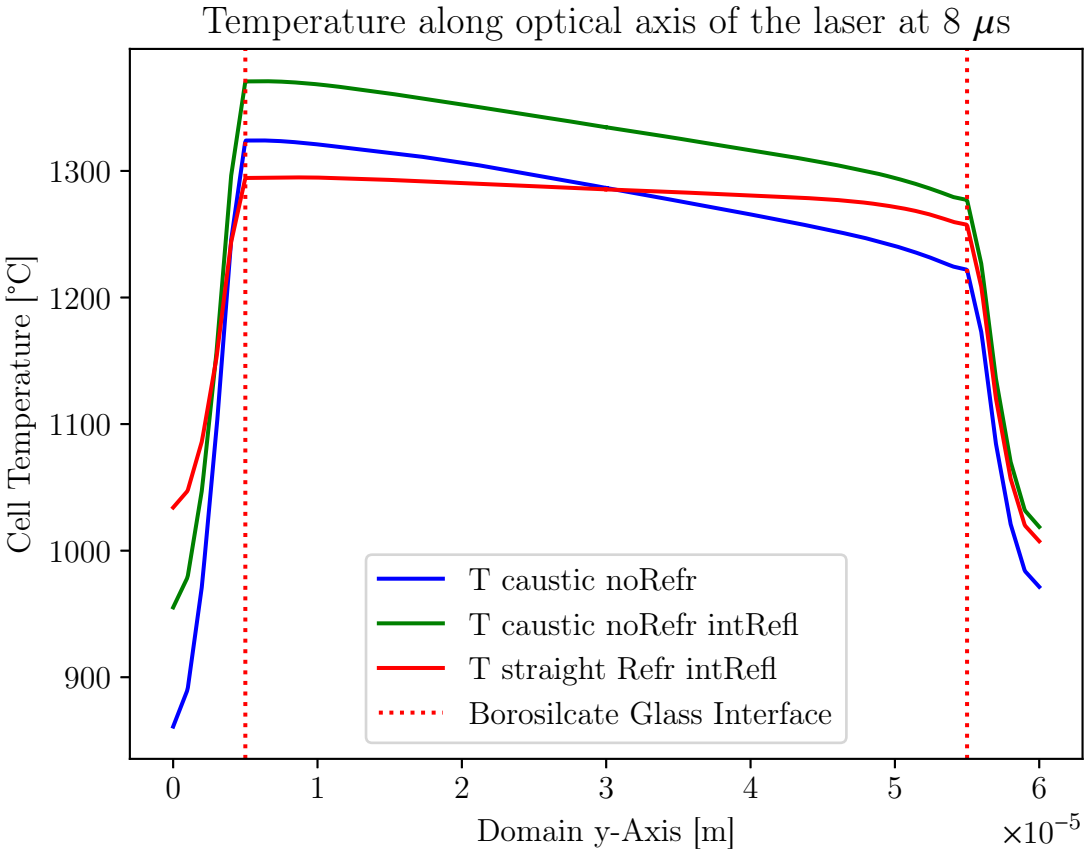

Figure 21: Temperature distribution through the work piece along the optical axis of the laser shows greatest heat generation at the bottom.

All simulations show the greatest temperature at the bottom of the glass. After 8 µs the work piece in case B has heated to around 1370 °C. Since the internal reflection at the bottom is considered, more laser energy was absorbed as in case A. The temperature gradient for both cases is larger compared to C. This is due to the refraction causing the beam to diverge as the particles are deflected towards the perpendicular of the interface. This creates a more even heat distribution, which counteracts the desired process.

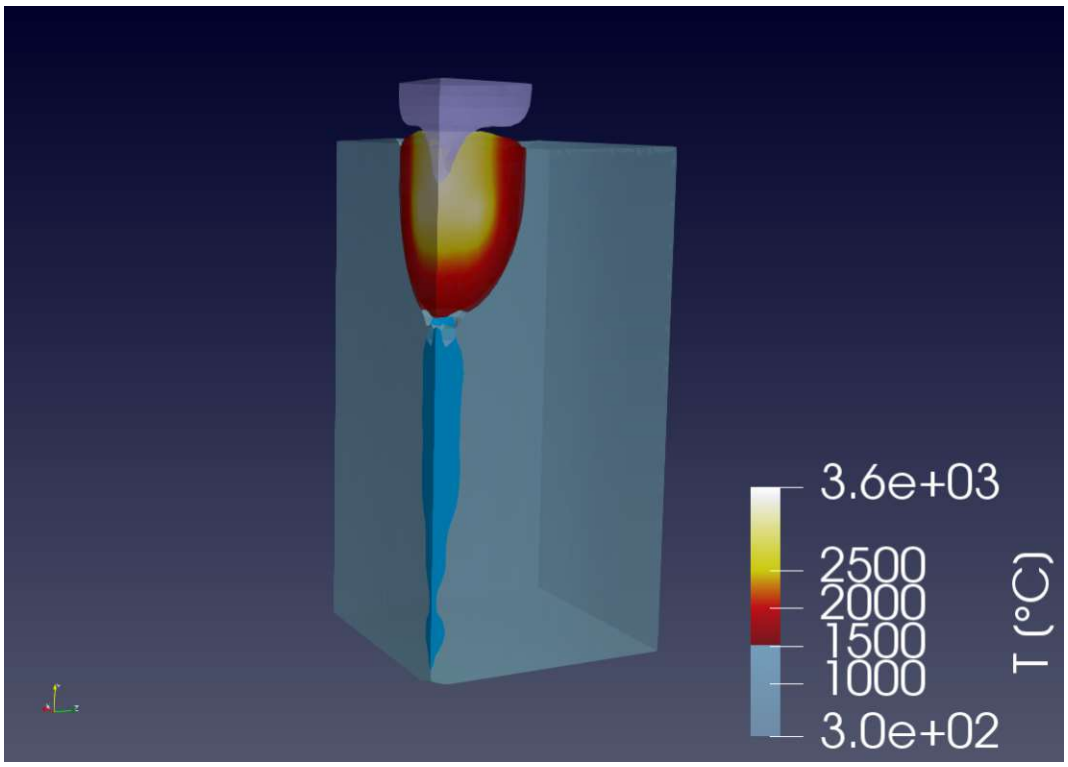

Figure 22: Case A: Evaporation begins at the top but the molten glass is solidified at the bottom.

Figure 22 shows the simulation process of case A after 31 µs. The glass has molten and resolidified at the bottom and mid section of the work piece. At the top surface, a keyhole is emerging. With the solid glass at the bottom, the vapor pressure cannot eject glass through the bore hole. The drilling process as described by Schrauben et al. [20] was not reproduced. In this simulation, the borehole is created solely by material vaporization.

Figure 23 shows the simulation of case B after 28 µs. Here, the evaporation at the top has just begun with the column of molten glass clearly visible throughout

the cross section of the work piece. At the sheath of the column, some cells show solidified melt. Some material is already pushed out from the underside of the bore hole. The simulation resembles the process as described by Schrauben et al. [20].

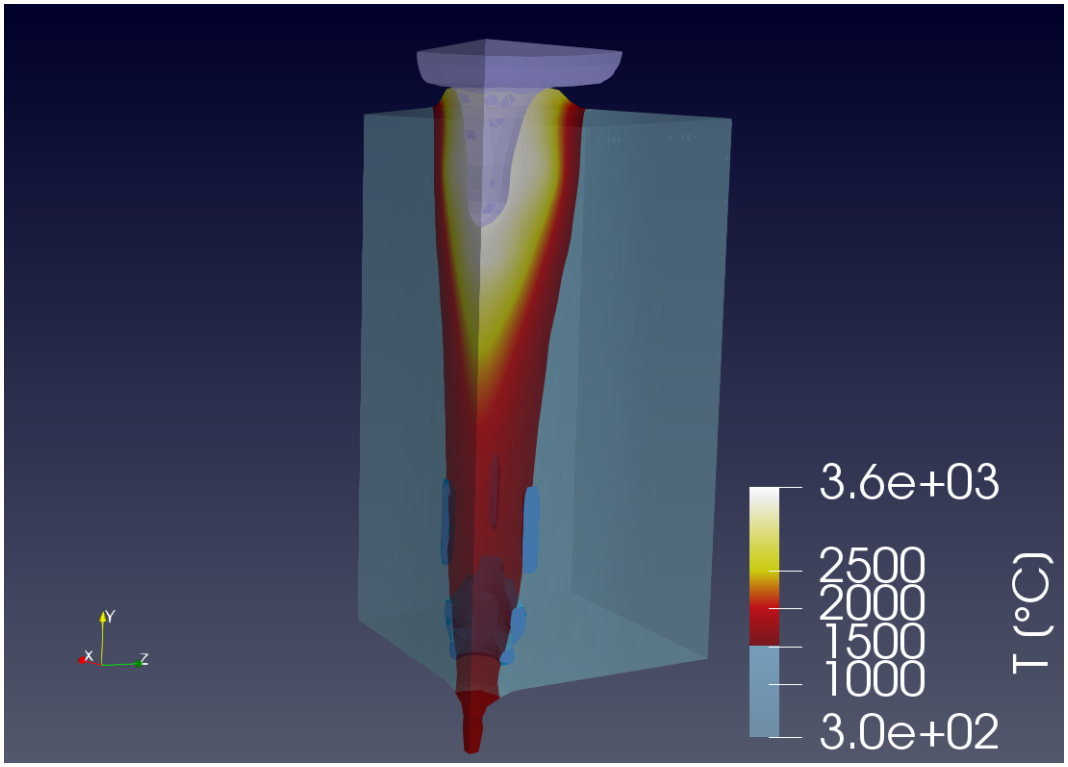

Figure 23: Case B: Evaporation begins at the top. The ejection of molten glass at the bottom is becoming apparent.

7 Conclusion

The main goal of this thesis was to accelerate an existing FVM-DEM solver by calculating the Lagrangian particle tracking on the GPU. More specifically, this involved the following tasks:

The 3D particle tracking algorithm and the photon particle interaction are implemented in CUDA to be executed on the GPU. A static CUDA library is compiled and linked to the solver. Each Eulerian timestep, the solver calls the GPU with all necessary particle and mesh field data. On return, the effects of the laser beam interaction are transferred to the OpenFOAM® mesh.

The existing photon parcel, particle and cloud classes are extended by the GPU functionality. Complying to the OpenFOAM® file structure to run applications, a flag can be set in the cloud dictionary file, whether the particle tracking should be executed on the CPU or GPU.

All written code is programmed in an object-oriented way. It was not necessary to comply to the template pattern idiom, as dynamic polymorphism with virtual functions was not used in the CUDA code. Static polymorphism in terms of base and derived classes was applied, as mentioned in chapter 5.

The thorough execution time analysis conducted in chapter 6 shows that speedups of factor 100 and more can be achieved compared to CPU execution. It is worth using the GPU accelerated solver if the number of particles is greater than 2000. Since undersampling the laser beam shows in the simulation results as symmetric stripes or patterns of uneven heat distribution, running simulations with so few particles does only make sense on very small domains.

The solver can be used to simulate laser based manufacturing processes like through hole drilling. Experimental results as described by Matsumoto et al. ([16], [20]) can be reproduced in terms of the heating propagation from the bottom to the top surface of the work piece. The ejection of molten glass from the bottom of the glass plate was shown.

8 Discussion and Future Work

Although it is possible to simulate realistic laser based manufacturing processes with the GPU accelerated solver, the current implementation comes with drawbacks. A software this size is always in a work-in-progress state. With its potential to massively accelerate the simulation process, it is worth discussing current limitations and the functionalities with the biggest prospective for improvement.

1. **MPI-GPU compatibility**
   OpenFOAM® supports the Message Passing Interface (MPI) implementation *openMPI* for running simulations in parallel on distributed processors. Large simulations are computed using domain decomposition, a method in which the geometry and associated fields are broken up and distributed on multiple processors for solution [11]. In the GPU-accelerated implementation only a single cloud object calling the kernel functions is supported. OpenFOAM®s domain decomposition will create one cloud object for each processor handling the particles in its respective part of the geometry. MPI-GPU compatibility can be achieved in two ways. Either by recomposing the separate clouds using a single master processor and subsequently calling the GPU kernel once for every time step. Alternatively, each processors cloud object could call a separate kernel for calculating the trajectories and interactions in their geometries. This demands for CUDA-aware *openMPI* and requires some changes in the OpenFOAM® source code.

2. **Phase interface evaluation and cell face evaluation**
   At the current state of the software, the boundary between any two phases is only evaluated at the cell face, i.e. the boundary between two cells. This is based on the erroneous assumption, that just a single phase can be present in any cell. Melting and evaporation cause phase advection into other cells. It is thus necessary to perform an interface evaluation after the particle tracking. Therefore, the phase fractions are interpolated to the previously obtained estimated end position or face intersection as well as the particles current position. In case of a phase conversion, the phase interface can be linearly

approximated. The particle is thereon moved onto the boundary between the two phases, where reflection and refraction are calculated.

3. **Kernel execution configuration**
   The main potential or advantage of calculating the laser beam propagation using particles instead of solving the radiative transport equation lies in the possibility to calculate multiple reflections and refractions with exact interface evaluations. Even modern ray tracing algorithms are not able to take all beam reflection effects into consideration. The kernel execution configuration is implemented in such a way, that one thread is called for each initially present particle at the beginning of each time step. Thus each thread calculates the trajectory and interaction of one particle and all its (through refraction) subsequently created children. Calculating more reflections and refractions for each photon particle will increase the quality of the simulation at the cost of exponentially higher computational costs. One idea to offset this increased execution time is to use more threads than the initial number of particles, with the redundant threads spinning on the memory location of the later added particles until they are created. Although this violates the principle of divergent branching mentioned in chapter 3.2, it could potentially speed up simulations with multiple refractions.

To conclude, the thesis has shown that FVM-DEM simulations can be accelerated by the use of GPUs. The particle tracking and interaction algorithm is parallelizable, enabling a faster and more accurate prediction of laser based manufacturing processes. Especially for scenarios where multiple reflections and refractions of the beam must be considered, the beam propagation using 'photonic' particles is worthwhile.

System Specifications

The benchmarks are conducted on a custom workstation. The system specifications can be seen below:

CPU Model:	i9-12900K
CPU Architecture:	Alder Lake
Total Cores:	16
Performance-cores:	8
Efficient-cores:	8
Total Threads:	24
Max Turbo Frequency:	5.20 GHz
Memory Type:	DDR5 4800 MT/s
Memory Size:	2 x 32 GB
Max Memory Bandwidth:	76.8 GB/sec
Cache:	30 MB
L2 Cache Size:	14 MB

Table 6: Workstation system specifications

GPU Model:	NVIDIA GeForce 3090
GPU Architecture:	NVIDIA Ampere
GPCs:	7
TPCs:	41
SMs:	82
CUDA Cores / SM:	128
CUDA Cores / GPU:	10469
Memory Clock (Data Rate):	19,5 Gbps
Memory Bandwidth:	936 GB/sec
L1 Data Cache/Shared Memory:	10469 KByte
L2 Cache Size:	6144 KByte

Table 7: GPU system specifications

Compilation & Documentation

The static `gpuTracking.so` library can be compiled using the provided `Allmake.sh` shell script or by executing the following `make` file:

Listing 1: Building gpuTracking.so

```
libgputracking.so: GPUTracking.cu ParticleData.cu
                   FlatMesh.cu ParticleEngine.cu
                   deviceManagement.cu Kernels.cu
        nvcc
        -I$(HOME)/NVIDIA_GPU_Computing_SDK/C/common/inc
        --compiler-options -Wextra,-Wall,-fPIC --shared
        -arch=sm_86 --ptxas-options=-v
        --linker-options -soname,libgputracking.so $^ -o $@

clean:
        rm -rf libgputracking.so
```

All code is documented using Doxygen documentation. The html files can be generated in the docs folder using the following command:

```
Doxygen doxyfile
```

The documentation can be compiled to PDF format with Latex in the respective folder using `make`.

Bibliography

[1] Nils Brünggel. „Lagrangian Particle Tracking on a GPU“. Diplomathesis. Lucerne University of Applied Sciences and Arts, 2011.

[2] M. Buttazzoni et al. „A Numerical Investigation of Laser Beam Welding of Stainless Steel Sheets with a Gap“. In: *Appl. Sci.* 11 (2021), p. 2549. DOI: https://doi.org/10.3390/app11062549.

[3] NVIDIA Corporation. *CUDA C++ Programming Guide*. Tech. rep. 2022.

[4] NVIDIA Corporation. *GPU Performace Background*. Tech. rep. 2022.

[5] NVIDIA Corporation. *NVIDIA Ampere GA102 GPU Architecture*. Tech. rep. 2021.

[6] Joshua Dyson. „GPU Accelerated Linear System Solvers for OpenFOAM and Their Application to Sprays“. Doctoral thesis. Brunel University London, 2017.

[7] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Publishers, 2015.

[8] The OpenFOAM Foundation. *OpenFOAM 6 Source Code*. URL: https://github.com/OpenFOAM/OpenFOAM-6. (accessed: 11.07.2022).

[9] Christopher Greenshields and Henry Weller. *Notes on Computational Fluid Dynamics: General Principles*. Reading, UK: CFD Direct Ltd, 2022.

[10] Christopher J. Greenshields and CFD Direct Ltd. *OpenFOAM Programmer's Guide*. OpenFOAM Foundation Ltd., 2015.

[11] Christopher J. Greenshields and CFD Direct Ltd. *OpenFOAM User Guide*. OpenFOAM Foundation Ltd., 2022.

[12] C.W Hirt and B.D Nichols. „Volume of fluid (VOF) method for the dynamics of free boundaries“. In: *Journal of Computational Physics* 39.1 (1981), pp. 201–225. DOI: https://doi.org/10.1016/0021-9991(81)90145-5.

[13] Helmut Hügel and Thomas Graf. *Laser in der Fertigung: Strahlquellen, Systeme, Fertigungsverfahren*. Wiesbaden: Vieweg+Teubner, 2009. DOI: 10.1007/978-3-8348-9570-7_1.

[14] Pijush Kundu, Ira Cohen, and David Dowling. *Fluid Mechanics*. Fifth. Elsevier Academic Press, 2012.

[15] Graham B. Macpherson, Niklas Nordin, and Henry G. Weller. „Particle tracking in unstructured, arbitrary polyhedral meshes for use in CFD and molecular dynamics“. In: *Communications in Numerical Methods in Engineering* (2008). DOI: 10.1002/cnm.1128.

[16] Hisashi Matsumoto et al. „Rapid formation of high aspect ratio through holes in thin glass substrates using an engineered, QCW laser approach“. In: *Applied Physics A* 128.4 (2022), pp. 1–10. DOI: https://doi.org/10.1007/s00339-022-05404-4.

[17] F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM® and Matlab*. Cham: Springer International Publishing, 2016. DOI: 10.1007/978-3-319-16874-6_1.

[18] A. Otto and R. Gómez Vázquez. „Fluid dynamical simulation of high speed micro welding“. In: *J. Laser Appl.* 30 (2018), p. 032411. DOI: https://doi.org/10.2351/1.5040652.

[19] Rupp, Karl. *42 Years of Microprocessor Trend Data*. [Online; accessed March 14, 2023]. 2017. URL: www.karlrupp.net.

[20] Joel N Schrauben et al. „Rapid and complex dynamics of through glass via formation using a picosecond quasi-continuous wave laser as revealed by time-resolved absorptance measurements and multiphase modeling“. In: *Applied Physics A* 129.4 (2023), p. 282.

[21] S. Whitaker. „Flow in porous media I: A theoretical derivation of Darcy's law“. In: *Transport in Porous Media* 1 (1986), pp. 3–25. DOI: https://doi.org/10.1007/BF01036523.