Speeding up proton therapy optimization algorithms using GPU-acceleration

S. Goudriaan

Speeding up proton therapy optimization algorithms using GPU-acceleration

by

S. Goudriaan

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Friday July 14, 2023 at 1:00 PM.

Student number:	5082978	
Project duration:	February 22, 2023 – July 14, 2023	
Thesis committee:	Prof. dr. ir. M. B. van Gijzen,	TU Delft, supervisor
	Dr. ir. D. Lathouwers,	TU Delft, supervisor
	Dr. P. M. Visser,	TU Delft
	Dr. Z. Perko,	TU Delft

TU Delft

Acknowledgements

I would like to thank my supervisors Martin van Gijzen and Danny Lathouwers for giving me the opportunity to take this unique and relevant assignment as my bachelor graduation project. The weekly meetings with them were of great value to my progress in the project. Their openness and patience during these meetings, and their amazing support throughout the project is greatly appreciated. Their sense of humor made the weekly meetings, and further communication via e-mail genuinely enjoyable. Furthermore I would like to thank my family for providing me with the opportunity to fully commit to my studies, without having to worry about anything else, and for all the support they have given me leading up to this moment.

S. Goudriaan
Delft, July 2023

Abstract

Proton irradiation therapy is a powerful form of cancer treatment, promising better dose conformity as compared to conventional radiotherapy. Due to the complex scattering properties of protons, the optimization process that is needed to accurately target the cancer cells whilst causing minimal damage to the surrounding healthy tissue and minimizing detrimental effects, is very time-consuming. This means that the CT scan it is based on has lost part of its accuracy in describing the to be irradiated tissue. To account for this, the surrounding tissue is irradiated more to ensure effective treatment, reducing dose conformity and thus increasing the probability of detrimental side effects.

To solve this problem, a new proton therapy method concept, called Online Adaptive Proton Therapy, or OAPT, calls for a CT scan to be taken about 30 seconds before the treatment, allowing the planned treatment to be adapted to any anatomical changes that have occurred since the previous scan where the original treatment planning was based on. The computations required for this adaptation, and the required quality assurance, currently still take significantly longer than 30 seconds on current computational hardware, making the concept not yet viable in its current form. However, using GPU acceleration, parts of the algorithm that is used for this can be significantly sped up to reduce overall computational time, potentially making the concept viable for real world application.

In this thesis, the research question *”How can GPU-offloading decrease computation time for proton therapy dose calculations?”* is answered by accelerating two model algorithms representative of two time-consuming steps in the proton therapy optimization process and analyzing the performance, on an NVIDIA V100S GPU, of the accelerated code. Furthermore, a model is postulated and validated to characterize and predict the performance of a GPU accelerated algorithm.

Using OpenACC, both algorithms achieved speedups between 30x and 440x excluding data transfer time, and between 0.88x and 40x including data transfer time, with both values depending on the problem size, with larger problems yielding larger speedups.

Further research is needed on the validity of the derived model on different hardware and for different algorithms. Furthermore, additional research on the effect of implementing these accelerated algorithms on the total computation time of the real world algorithm is advised.

Contents

1 Introduction **1**
1.1 Proton therapy . . . 1
1.1.1 How it works . . . 1
1.1.2 Advantages over conventional radiotherapy . . . 1
1.1.3 State of the art . . . 2
1.2 Scope of this thesis . . . 3
1.2.1 Research question . . . 3
1.2.2 General approach and restrictions . . . 3
1.2.3 Outline of the report . . . 3

2 The proton transport problem **4**
2.1 Physical background . . . 4
2.1.1 The linear Boltzmann equation . . . 4
2.1.2 The Fokker-Planck equation . . . 5
2.1.3 Boundary conditions . . . 5
2.2 The numerical algorithm . . . 6
2.2.1 Numerical method and discretization . . . 6
2.2.2 matvec . . . 7
2.2.3 Plane sweep . . . 7

3 Parallel computing **9**
3.1 Scientific computing in Fortran . . . 9
3.1.1 General structure . . . 9
3.1.2 Available compilers . . . 9
3.1.3 Parallel computing support . . . 9
3.2 CPU vs GPU based programming . . . 10
3.2.1 Architectural differences . . . 10
3.2.2 Differences in performance scaling . . . 10
3.3 GPU offloading implementation . . . 11
3.3.1 Standard parallel: do concurrent . . . 11
3.3.2 OpenACC . . . 11
3.3.3 OpenMP . . . 12
3.3.4 NVIDIA CUDA Fortran . . . 12
3.3.5 Trade-off . . . 12
3.4 Modeling GPU-acceleration potential . . . 12
3.4.1 Available parallelism and execution time . . . 12
3.4.2 Data transfer time . . . 14
3.4.3 Miscellaneous factors . . . 15

4 Experimental method **16**
4.1 Acceleration process . . . 16
4.1.1 Directives and clauses . . . 16
4.1.2 Data movement optimization . . . 17
4.1.3 Mapping threads across levels of parallelism . . . 17
4.1.4 Exposing more parallelism . . . 18
4.1.5 Further optimizations . . . 18
4.1.6 data representation . . . 18

4.2 Performance measurements . . . 19
4.2.1 Hardware . . . 19
4.2.2 Compilers . . . 19
4.2.3 Compiler options . . . 19
4.2.4 Code segments . . . 21
4.2.5 System clock . . . 22
4.2.6 Profiling tools . . . 22

5 Results and discussion 23
5.1 Plane sweep . . . 23
5.1.1 Pinned memory . . . 25
5.1.2 Multicore . . . 25
5.1.3 Data representation . . . 26
5.1.4 Gfortran and ifx . . . 27
5.1.5 Correctness . . . 27
5.2 Matvec . . . 28
5.3 General implementation considerations . . . 29
5.4 Applying the general performance model . . . 30
5.4.1 Execution time . . . 30
5.4.2 Data transfer time . . . 34
5.4.3 Discussion of the model . . . 35

6 Conclusion 36
6.1 The performance model . . . 36
6.2 Acceleration results . . . 37

A Model code 38
A.1 Matvec model code . . . 38
A.2 Plane sweep model code . . . 44

B Accelerated code 51
B.1 Accelerated matvec model code . . . 51
B.2 Accelerated plane sweep model code . . . 57

C Performance measurement tables 67
C.1 Matvec . . . 67
C.2 Plane sweep . . . 69

1 Introduction

1.1. Proton therapy

In the economically developed world, cancer is the leading cause of death, and has been for the past decades. New methods are constantly being developed and improved to cure this disease. Proton irradiation therapy, often referred to as just ”proton therapy”, is a form of cancer irradiation therapy (often referred to as radiotherapy), where cancerous tumors are bombarded with ionizing radiation to break apart cancerous cells (Uilkema, 2012). In this section, the concept of proton therapy will be briefly introduced. The section will be concluded with a short analysis of the current state-of-the-art in proton therapy, as well as the problem this thesis aims to help solve some issues with the current methods applied in this field.

1.1.1. How it works

As introduced earlier, proton therapy is a form of cancer irradiation therapy. As the name implies, proton therapy makes use of protons, which are positively charged subatomic particles. It uses those protons to irradiate the cancerous tumor, with the aim of breaking the cancerous cells apart. In a proton therapy clinic, protons are accelerated in a small particle accelerator and carefully delivered to a targeted area in the patient.

As any form of ionizing radiation, protons can just as easily damage healthy cells as it can damage cancer cells. As damaging too many healthy cells can cause undesired side effects, an important consideration when planning any form of radiation therapy, is minimizing this damage to healthy cells.

The amount of damage caused by ionizing radiation in a given area directly correlates with the energy deposition in that area. Therefore, this energy deposition is used as a measure for damage caused. In irradiation therapy, treatment planning is an optimization process, where, in general, the energy deposition is to be simultaneously maximized inside the tumor, while being minimized in the healthy tissue and vital organs around it (Levin et al., 2005).

1.1.2. Advantages over conventional radiotherapy

The fundamental difference between proton therapy and conventional radiotherapy, is the type of radiation used. Conventional radiotherapy uses photons, which have a very long range, depositing energy relatively evenly spread over their trajectory. Proton therapy, as introduced earlier, uses protons, which have a finite range and deposit most of their energy at the end of their range. The range of a proton depends on the medium it travels to, and the initial kinetic energy supplied to the proton. Proton therapy utilizes those properties to more precisely control where the energy is deposited.

In figure 1.1, the relative delivered dose of proton therapy is compared with the relative dose in conventional radiation therapy, as function of penetration depth. The area between the vertical dotted lines is the irradiated tumor. The black curve depicts the relative energy deposition using photons, the blue curves depict the dose distribution of protons at different energy levels, and the red curve depicts the superposition of the individually modulated protons energy deposition. As can be seen in the figure, the relative dose delivered in the tumor region by the protons is significantly higher, as compared to the

photons, whilst the opposite is true for the range outside the tumor, where healthy tissue is situated. This gives proton therapy a significant advantage regarding *dose conformity*, which means that more of the total dose is delivered in the tumor, sparing the healty tissue and critical organs around it.

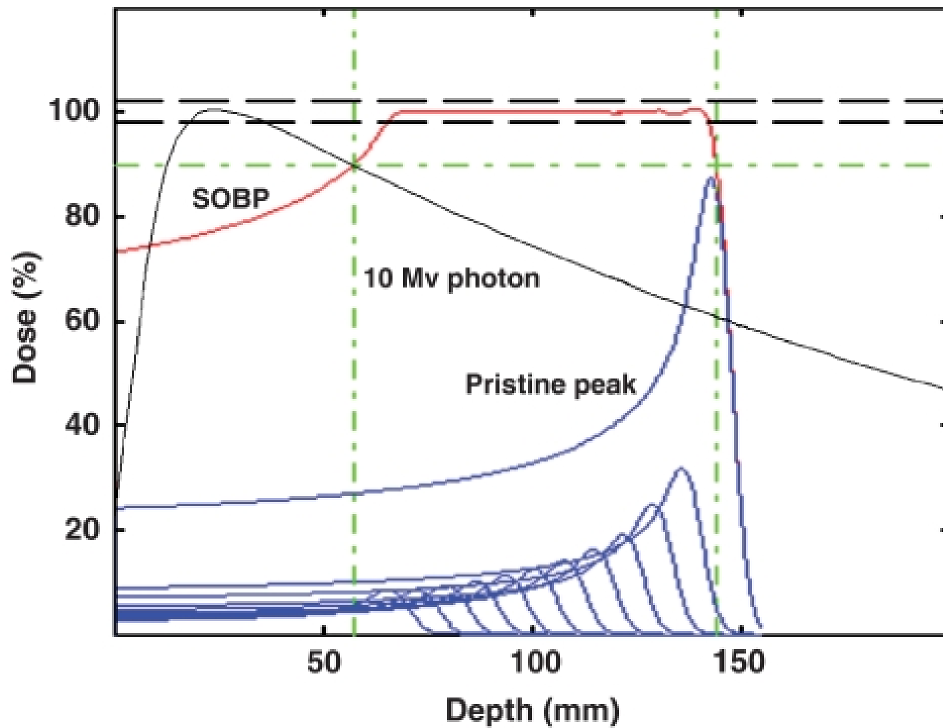


Figure 1.1: Relative dose distribution obtained using photons, compared to the relative dose distribution of proton irradiation therapy, as function of depth. The area in between the vertical dotted lines is the irradiated tumor. This figure was extracted from Levin et al., 2005.

1.1.3. State of the art

In order to optimize the dose conformity as mentioned earlier, a special CT scan is needed to determine the exact position of the tumor, as well as critical organs, and to model the scattering properties of the medium. Those scattering properties are needed to accurately determine the behavior and energy deposition of the protons, as elaborated in section 2.1. As stated in a recent paper by Burlacu et al., 2023, the optimization process is currently very long, meaning that anatomical varieties, such as weight loss, which are to be expected over the course of the often weeks long treatment process, are likely to have a significant impact on the accuracy of the delivered dose. To account for this in current state-of-the-art therapy planning, the dose distribution is robustly optimized to guarantee a sufficient dose is delivered in the tumor. A consequence of this, however, is that the surrounding tissue is still damaged more than ideal, increasing the likelihood of detrimental side effects.
A more ideal approach would be to implement a concept called ”Online Adaptive Proton Therapy”, or OAPT for short, where a CT scan is made approximately 30 seconds before the protons are delivered, such that the therapy plan can be adapted to any anatomical changes that occurred before the scan took place. This increases the accuracy of the anatomical model as compared to the actual tissue during the session.
However, as stated in Burlacu et al., 2023, currently the computations required to re-optimize the planned treatment to account for those anatomical variations, and the required quality assurance process, take far longer than 30 seconds. To overcome this issue, a deterministic algorithm is presented in the paper. Running sequentially, that is on a single CPU core, this algorithm still takes too long to complete. However, if some time-consuming steps in this algorithm can be accelerated using GPU offloading, this algorithm may be viable for use in the quality assurance and re-optimization steps.

1.2. Scope of this thesis

In this section, the scope of this thesis is defined, introducing the research question, and summarizing the approach taken to answer it, as well as summarizing the relevant restrictions put in place to control the scope of the thesis. At the end of this section, an outline of the report is provided, summarizing the scope of each individual chapter.

1.2.1. Research question

The following question is central to this thesis:
"How can GPU-offloading decrease computation time for proton therapy dose calculations?"
This research question has two separate interpretations that are both relevant to this thesis. The first interpretation asks for the **methods** GPU offloading can be implemented to decrease computation time, whilst the second interpretation asks for how **much** computation time can be reduced using GPU offloading. This thesis aims to answer both questions at the same time, by accelerating two pieces of model code that are representative of two of the main ingredients of the algorithm used in those calculations, documenting the acceleration process, analyzing the performance of the accelerated code, and developing a model to predict the performance of similar algorithms.

1.2.2. General approach and restrictions

The starting point of this thesis is a set of two model algorithms designed to execute one step each in the linear Boltzmann solver that is used in the proton therapy dose calculation and optimization process. The aim is to find the following three things:

1. **A method** of accelerating the numerical algorithms in Fortran.
2. **A measure** of the performance improvements accelerating those algorithm brings.
3. **A model** to predict the accelerated performance of similar algorithms.

Each item in this list is a step in completing the next item on the list. Therefore the general approach of this thesis is to compare multiple available methods, choose the one that best fits the application, implement the chosen method, test and evaluate its performance, and derive and validate a general model to describe its performance.
Because this thesis is to be completed in a limited time frame, the scope is subject to the following restrictions:

1. Not all available methods can be investigated. There are many different ways to implement GPU acceleration, with only limited time to pick one of them.
2. The algorithms to be accelerated, are simplified model problems, which are still representative of the algorithms in the real world application. Implementation in the real world algorithm requires a few more adaptations, which are outside the scope of this thesis.
3. The performance of the accelerated code is only tested on a single hardware configuration. However, performance portability is still considered when choosing the method of acceleration.
4. Ease of implementation is an important consideration in choosing the acceleration method.

1.2.3. Outline of the report

Now that the scope of this thesis is defined, it is time to outline the contents of this report.
In the next chapter, the physical and numerical problem relevant to the investigated algorithms are explored. Afterwards, the relevant concepts about GPU acceleration are introduced, as well as the methods that were considered to achieve this in chapter 3. In the same chapter, a general model is postulated to characterize the performance of GPU accelerated code. In the chapter after that, the acceleration process, the method of measuring performance, and the hardware the (accelerated) code is tested on are introduced. In chapter 5, the results of those measurements are presented, the model is applied and evaluated, and the validity of the results is discussed. Finally, the thesis will be closed off with a conclusion summarizing the results and providing a few suggestions for future research.

2 The proton transport problem

As stated in the previous chapter, this thesis aims to determine the GPU-acceleration potential of an algorithm used to solve proton transport problems for proton therapy applications. In this chapter, the physical background of this transport problem is explained in more detail, as well as a more in-depth analysis of the algorithm of interest.

2.1. Physical background

Transport of heavy charged particles, and protons in particular, is relevant in many important applications (Zheng-Ming & Brahme, 1993). However, this thesis will focus on one specific medical application: proton irradiation therapy for cancer treatment. As briefly described in section 1.1.1, proton therapy works by irradiating a targeted tissue with protons, which deposit their energy, and therefore cause the most damage, in a narrow region near the end of their trajectory. To minimize the damage done to healthy tissue, and maximize the damage in the tumor, the trajectory and energy deposition of the protons must be accurately modeled.
The most accurate method for modeling this energy deposition, is based on simulating each proton individually and the probabilistic paths it could follow due to its interactions with the irradiated tissue. However, as these simulations, also known as Monte Carlo simulations, are very computationally expensive, this method is not feasible for use in proton therapy planning. Therefore a deterministic approach is used for this purpose. This approach is described in more detail later in this chapter, but first the background of the proton transport problem, and the associated equations, is explained.

2.1.1. The linear Boltzmann equation

A central equation to particle transport problems is the linear Boltzmann equation. The Boltzmann equation governs how (on average) the protons scatter as a result from their interactions with the medium they travel through, as well as the resulting energy deposition. Its validity for proton therapy applications is supported in chapter 3 of "Proton Therapy Planning using the S_N Method with the Fokker-Planck Approximation" by Uilkema, 2012. Leaving out the time dependent variable, which is not relevant for proton therapy as only the steady-state solution is relevant here, the linear form of the steady state Boltzmann equation reads

$$\hat{\Omega} \cdot \nabla \phi(\vec{r}, E, \hat{\Omega}) + \sigma_t \phi(\vec{r}, E, \hat{\Omega}) = \int_{4\pi} \int_0^\infty \sigma_s(\vec{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega}) \phi(\vec{r}, E', \hat{\Omega}') dE' d\hat{\Omega}'. \tag{2.1}$$

Here $\phi(\vec{r}, E, \hat{\Omega})$ denotes the angular flux at position $\vec{r}$ of particles of energy E traveling along unit vector $\hat{\Omega}$, σ_t is the total scattering cross section and $\sigma_s(\vec{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega})$ being the differential cross section of particles at position $\vec{r}$ scattering from energy state E' to E and change direction from $\hat{\Omega}'$ to $\hat{\Omega}$ simultaneously. The derivation of this equation is beyond the scope of this thesis and can be found in chapter four of "Nuclear reactor analysis" by Duderstadt and Hamilton, 1976. As further elaborated by Uilkema, 2012, as well as by Duderstadt and Hamilton, 1976, the first term of equation 2.1 is the streaming term, describing the free movement of the particles through the domain, the second term

describes the particles which change direction and/or energy due to all forms of scattering, and the integral term describes all particles scattering from other (higher) energies and different directions to energy state E and direction $\hat{\Omega}$, at $\vec{r}$. This third term is often referred to as the "Boltzmann scatter operator".
Solving equation 2.1 would deliver all the information needed to calculate both the average trajectory and energy deposition of the particles. Due to the near-singular shape of σ_s in the angular domain for charged particles, however, as shown by Uilkema, 2012, things are not as easy when dealing with protons. This is because deterministic numerical methods would commonly need to use (Legendre) polynomials to approximate σ_s, which need very high orders of polynomials to approximate such a steep function with any degree of accuracy. This, together with the very finely discretized angular and energetic grid needed to simulate the many collisions accurately, makes directly solving this equation in its current form unpractical for application in the field of proton therapy (and many other fields). Therefore an approximation is needed: the Fokker-Planck equation.

2.1.2. The Fokker-Planck equation

To arrive at this approximate form of equation 2.1, the first step is to split the Boltzmann scatter operator and with it the two cross sections in three parts: σ_a for absorption, i.e. proton is absorbed by the nucleus, σ_e for elastic scatter (no kinetic energy lost), and σ_{in} for inelastic scatter. The equation now has the following form:

$$\begin{aligned}\hat{\Omega}\cdot\nabla\phi(\vec{r},E,\hat{\Omega}) = &\int_{4\pi}\int_0^\infty \sigma_{s,a}(\vec{r},E'\to E,\hat{\Omega}'\to\hat{\Omega})\phi(\vec{r},E',\hat{\Omega}')dE'd\hat{\Omega}' - \sigma_{t,a}\phi(\vec{r},E,\hat{\Omega}) \\ &+\int_{4\pi}\sigma_{s,e}(\vec{r},E,\hat{\Omega}'\to\hat{\Omega})\phi(\vec{r},E,\hat{\Omega}')d\hat{\Omega}' - \sigma_{t,e}\phi(\vec{r},E,\hat{\Omega}) \\ &+\int_0^\infty \sigma_{s,in}(\vec{r},E+Q\to E,\hat{\Omega})\phi(\vec{r},E+Q,\hat{\Omega})dQ - \sigma_{t,in}\phi(\vec{r},E,\hat{\Omega}).\end{aligned} \tag{2.2}$$

Here, the assumption is made explicit that inelastic scatter cannot increase the energy of the colliding proton by redefining E' as $E+Q$, where Q denotes the energy transferred from the proton to the medium. Another assumption that is made already, is that in case of elastic scatter the energy transfer is negligible, as the protons have a much higher mass as compared to the electrons they interact with (Burlacu et al., 2023).
The first integral, with $\sigma_{s,a}$, can be neglected completely, the effect of the second integral is can be expressed as a source term $S(\vec{r},\phi)$[1], and the third integral, which accounts for a phenomenon called "energy straggling" is neglected for scattered protons. Unscattered protons are handled by a separate ray-tracing algorithm which is outside the scope of this thesis.[2] The equation is solved at a single energy level (discretization of the energy domain is outside the scope of this thesis), further simplifying the equation that is solved. This leads us to the equation that is solved by the algorithm relevant to this thesis:

$$\hat{\Omega}\cdot\nabla\phi_s(\vec{r},\hat{\Omega}) + \sigma_t\phi_s(\vec{r},\hat{\Omega}) = S(\vec{r},\phi_s), \tag{2.3}$$

where $S(\vec{r},\phi)$ is a source term, representing for instance protons that are "scattered in" from different directions or higher energy levels, and ϕ_s is the flux of scattering protons. This equation is a further simplified version of what can be found in Lathouwers, 2023, but is sufficient for the scope of this thesis.

2.1.3. Boundary conditions

So far the boundary conditions have been left open. For the purposes of this thesis, those are given by:

$$\phi(\vec{r}_s,\hat{\Omega}) = \begin{cases}\phi_{in}(\vec{r}_s), & if\, \hat{\Omega}\cdot\vec{e}_s < 0 \\ 0, & otherwise\end{cases} \forall\vec{r}_s\in\partial V. \tag{2.4}$$

Here, ∂V refers to the boundaries of the domain, and $\phi_{in}(\vec{r}_s)$ is a given function denoting the inflow at the boundaries. This boundary condition is of the non-reentrant type as found in Burlacu et al., 2023.

[1]This source term is also outside of the scope of this thesis.
[2]The reason for this separation, is that unscattered protons are concentrated in a very narrow region, making it hard for most numerical methods to accurately model them together with the scattered protons.

2.2. The numerical algorithm

Now that we have arrived at the equation that is to be solved for the model problem, it is time to discuss how it is solved numerically.

2.2.1. Numerical method and discretization

The first step in solving any equation numerically, is choosing which method to use and discretizing the equation. The method of choice here is discontinuous Galerkin for spatial discretization, and discrete ordinates for angular discretization. The working of this method will be be explained later in this subsection. Applying angular discretization on equation 2.3, the following equation is obtained:

$$\hat{\Omega}_n \cdot \nabla\phi_n(\vec{r},\hat{\Omega}) + \sigma_t\phi_s(\vec{r},\hat{\Omega}) = S(\vec{r},\phi_n), \tag{2.5}$$

where $\hat{\Omega}_n$ is the nth ordinate of the discretized direction vector space, with ϕ_n the associated flux. The source term is now defined by some polynomial for each cell and direction. In the case of the test algorithm, the source term is set to zero for each cell and direction, but this is not a strict requirement for the real world implementation. Also note that there still is one more step required to fully discretize this equation: the spatial discretization, and with that the gradient of ϕ_n, which is still denoted by the ∇ operator, which needs to be implemented numerically. However, this will be touched upon later in this section, as first the numerical method need to be introduced in more detail.

Discontinuous Galerkin method

The numerical method used for this thesis, is the first order discontinuous Galerkin method, also known as dG1. This method is a mix of the finite volume and finite element method, approximating the solution using a first order polynomial for each cell. To solve the system of equations, it does need the values of the surrounding cells as boundary conditions, making the cells not fully independent. As elaborated in subsection 2.2.3 however, there is a way to work around that to still allow good parallelism.
The exact formulation of dG1 is outside of the scope of this thesis. Instead, the reader is encouraged to read more about the general concept of this method in ”Mathematical Aspects of Discontinuous Galerkin Methods” by Di Pietro and Ern, 2011.

Polynomial basis and the cell-level system of equations

The constants defining the aforementioned polynomial are expressed in vector form, relating to the original polynomial through a basis. The basis used the model code is of the form $\{1, x, y, z\}$, reducing the first order polynomials to vectors of length four. $\phi_n[e]$ For instance becomes $\phi_{n,e}(a, b, c, d)$, where a, b, c, d refer to the coefficients of some polynomial.[3]
The first step to find the system of equations to be solved for each cell is to multiply each term by h_j and integrating over the volume of the cell:

$$\int_e h_j\hat{\Omega}\cdot\nabla\phi d\vec{r} + \int_e \sigma_t\phi h_j d\vec{r} = \int_e S h_j d\vec{r}, \text{ for } h_j \in \{1, x, y, z\}. \tag{2.6}$$

Here h_j corresponds to the elements of the polynomial base. In order to discretize equation 2.6, consider the three integrals separately:

$$\int_e h_j\hat{\Omega}\cdot\nabla\phi d\vec{r} = -\int_e \phi(\nabla h_j\cdot\hat{\Omega})d\vec{r} + \int_e \hat{\Omega}\cdot\nabla(h_j\phi)d\vec{r} \tag{2.7}$$

$$\int_e \sigma_t\phi h_j d\vec{r} = \sigma_t\sum_i \phi_e^i\int_e h_i h_j d\vec{r} \tag{2.8}$$

$$\int_e S h_j d\vec{r} = 0 \qquad \text{(given for the test case)} \tag{2.9}$$

Equation 2.7 stems from integration by parts, and equation 2.8 stems from the identity $\phi = \sum_i \phi_e^i h_i$. The second equation reduces further to ϕ_e^1 in the case $h_j = 1$ and to $\frac{1}{3}\phi_e^j$ in the case $h_j \in \{x, y, z\}$. ϕ_e^i

[3]Each e in $\phi_n[e]$ referenced earlier actually refers to a set of four elements in the vector $\vec{\phi}_n$ in the computational scheme. For the definition of the problem and its solution, this is however irrelevant.

here refers to the ith constant defining the polynomial approximation of ϕ in element e.
Further evaluation of equation 2.7 finally reduces the first integral to:

$$2\phi_e^1 \cdot \begin{cases} 0, & if h_j = 1 \\ \Omega_1, & if h_j = x, \\ \Omega_2, & if h_j = y, \\ \Omega_3, & if h_j = z, \end{cases} \tag{2.10}$$

where Ω_i refers to the ith index of $\hat{\Omega}$. Using the following relation for the final integral:

$$\int_e \hat{\Omega} \cdot \nabla(h_j \phi) d\vec{r} = \int_{\partial e} h_j \phi (\hat{\Omega} \cdot \hat{n}) d\vec{r}, \tag{2.11}$$

together with the fact that ϕ at the upstream boundary of a cell is determined by its neighboring cell (protons don't just disappear or appear at a boundary between cells), equation 2.6 is finally ready to be put in matrix form $\mathbf{A}\vec{x} = \vec{b}$ to be solved using the Krylov method. The exact workings of this Krylov method, and the exact form of the matrix partially constructed above, are outside the scope of this thesis. For the purposes of this thesis, it is sufficient to know that the Krylov method works using moments of vector $\vec{x}$ in matrix **A**. The nth moment of a vector $\vec{x}$ in a matrix is given by this vector multiplied with matrix **A**, repeated n times: $\mathbf{A}^n \vec{x}$. The next subsections will introduce the main ingredient to this method, as well as a preconditioner to help it work efficiently, as those two parts of the algorithm were accelerated during this project.

2.2.2. matvec

The matvec algorithm is the main ingredient to the application of the Krylov method. For each element in the grid, it constructs the 4x4 matrix **A** as mentioned earlier, and defined in more detail in the next subsection, and multiplies it with a given vector $\vec{\phi}$. It then subtracts it from a different vector constructed from the $\vec{\phi}$ given for its upstream neighbors. It adds the result to another given vector for its element and with that completes the algorithm. The model code used for this algorithm can be found in appendix A.1. This algorithm is relatively memory intensive and its elements are fully independent, meaning that it is fully parallel.

2.2.3. Plane sweep

The plane sweep algorithm essentially solves a system $\mathbf{A}\vec{x} = \vec{b}$ for each element by sweeping through the domain along the $\hat{\Omega}$ vector, evaluating every cell in a "parallel plane" (as defined later in this subsection) before moving to the next.[4] A full description of the role and working of this algorithm can be found in Kópházi and Lathouwers, 2015, and the model code used for this algorithm is provided in appendix A.2. In this subsection, a brief summary of the functionality of this algorithm is given, as well as a more in-depth analysis of its structure. In the summary given, it is assumed that $\hat{\Omega}$ is in the positive octant, that is, the direction of flow is in the direction of positive x, y and z. The same algorithm works the same way in different octants, with slightly different indices.
For each discrete direction $\hat{\Omega}_n$ and element e, the plane sweep algorithm solves the following equation:

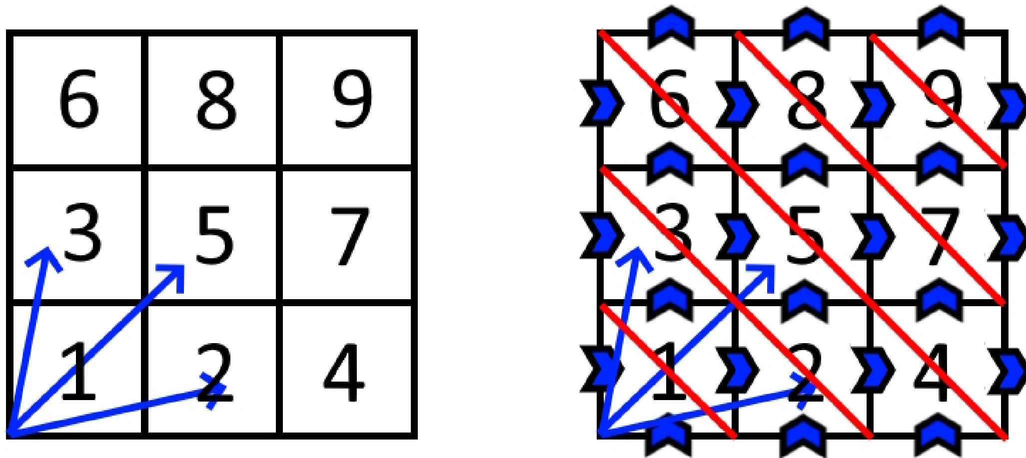
$$\mathbf{A}(\hat{\Omega})\vec{\phi}_{e(i,j,k)} = \Omega_1 \mathbf{M}_{xm} \vec{\phi}_{e(i-1,j,k)} + \Omega_2 \mathbf{M}_{ym} \vec{\phi}_{e(i,j-1,k)} + \Omega_3 \mathbf{M}_{zm} \vec{\phi}_{e(i,j,k-1)}. \tag{2.12}$$

In this equation, all bold faced capital letters denote 4x4 matrices and all vectors have a length of four. In this equation, $\vec{\phi}_{e(i,j,k)}$ is the vector of polynomial constants, as defined in a previous section, in the element with spatial coordinates (x_i, y_j, z_k). Matrix $\mathbf{A}(\hat{\Omega})$ is built from the following constants and sparse matrices (and the $\hat{\Omega}$ vector):

$$\mathbf{A}(\hat{\Omega}) = \hat{\Omega} \cdot \mathbf{K} + \Omega_1 \mathbf{M}_{xp} + \Omega_2 \mathbf{M}_{yp} + \Omega_3 \mathbf{M}_{zp} + \sigma_t \mathbf{M}. \tag{2.13}$$

[4]In the case of the model problem, with S=0, the plane sweep algorithm can solve the problem directly. However, because the plane sweep algorithm assumes that all directions and energy levels are fully independent, it cannot take into account particles that scatter from higher energy levels or different ordinates. This is why for the real world application, a Krylov method is used, where both algorithms work in conjunction to solve the proton transport problem.

All other matrices in 2.12 are sparse. The exact values in these matrices are not relevant for now.
For the purpose of this thesis, the parallel structure of this algorithm is the most relevant. As can be seen from equation 2.12, the system the plane sweep algorithm solves to find $\vec{\phi}_{e(i,j,k)}$, depends on $\vec{\phi}_{e(i-1,j,k)}$, $\vec{\phi}_{e(i,j-1,k)}$ and $\vec{\phi}_{e(i,j,k-1)}$, which are the $\vec{\phi}_{e(i,j,k)}$ for its upstream neighbors. This means that the algorithm first has to solve the system for its upstream neighbors before it can solve the system of a given element. Therefore, there is a strict order in which the elements must be evaluated. However, as the boundary conditions are of the non-reentrant type, and the modeled flow is along a given vector, the domain can be divided up into planes of parallelism, with each element in a given plane being independent from any other element in the same plane. This principle is illustrated in 2D in figures 2.1a and 2.1b, but works in the same way as in 3D, as illustrated in figure 2.2. The main conceptual difference here is that the planes of parallelism in 3D are lines in the 2D variant.



(a) Example of a 3^2 grid with three arbitrary vectors in in the first quadrant.

(b) Example of a 3^2 grid, now with the direction of flow between the cells and parallel "planes".

As becomes clear from 2.1a, following any vector in this quadrant, it is impossible to move from cell 2 to cell 3. In figure 2.1b, the vectors are broken down in a horizontal and vertical component (not to scale) and placed on the cell borders to show how they affect flow from cell to cell. It becomes clear that it is impossible to move between cells on the same red line, demonstrating the mutual independence between those cells. The red lines are the 2D equivalent of the planes that are used in our 3D model. In the 2D example, the number of cells in each parallel "plane" is, from bottom left to top right, {1,2,3,2,1}. For the 3D case this is {1, 3, 6, 7, 6, 3, 1}. Until the planes pass one of the corner points, the number of elements increases quadratically, as demonstrated in section 5.4.1.

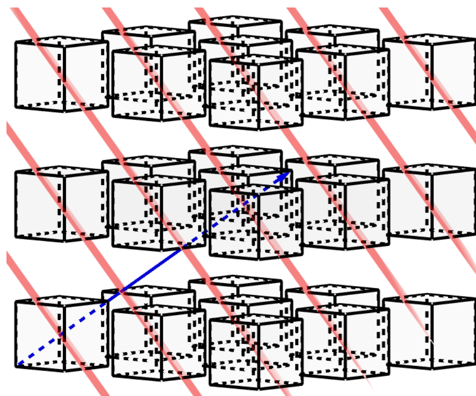


Figure 2.2: Example of parallel planes in a 3^3 cube grid. Here the elements in the grid are exploded for clarity. The image is projected in parallel to the planes, making them show up as lines.

3
Parallel computing

In this chapter, the basic programming concepts that are relevant for this thesis will be introduced. Afterwards the hardware of CPUs and GPUs is compared from a programming perspective. Then the implementation of GPU offloading is discussed, after which a general GPU performance model will be postulated.

3.1. Scientific computing in Fortran

The algorithms that are accelerated for this thesis are provided in Fortran. Fortran is one of the original compiled languages and is specifically developed for scientific computing. With *compiled language*, it is meant that the *source code*, which is the program as written by the programmer, as a whole, is *compiled* by a specialized *compiler* to generate an executable file with machine instructions for the computer hardware. These instructions are often specific to a certain hardware configuration, which is either the configuration it was compiled on, or a different hardware set as defined by the user using *compiler options*. Compiler options are additional instructions provided to the compiler at *compile time*, which is the moment the compiler compiles the source code. In this section, the programming language and its use for this thesis is explained in more detail ("The Fortran Programming Language", 2022).

3.1.1. General structure

A Fortran source code file generally consists of three parts: Variable declaration, the main program, and the subroutines and functions.
In the first part, all variables are declared. These variables are either declared as allocatable, or allocated directly. If a variable is allocated directly, the size and shape are already declared, otherwise this happens later when it is allocated. If external modules are used, they are loaded at the very beginning of the source code.
In the second part, the main program, functions and subroutines can be called to modify the variables and perform all sorts of tasks.
In the third part, all the (sub-) routines and functions are declared. These routines and functions can also be loaded from a separate module at the start of the program.

3.1.2. Available compilers

There are many compilers available for use with the Fortran standard. Each compiler is developed with a different implementation and different hardware in mind. This means that the compiler choice greatly impacts the performance on a given set of hardware. The difference in performance that was found during this thesis, on the hardware as specified in section 4.2.1, will be discussed in section 5.1.4 and section 5.2. The compilers that are used for this thesis are listed and compared in section 4.2.2.

3.1.3. Parallel computing support

In 2008, a revision of the standard was released, with built-in support for parallel computing with the use of do concurrent and co-arrays. Many modern compilers also support parallel computing through compiler directives. This thesis focuses on GPU acceleration. Section 3.3 goes more in depth of the

specific considerations that go into choosing which method to use when implementing GPU offloading for existing code.

3.2. CPU vs GPU based programming

When writing any program, it is important to take into account what hardware it should run on. Because CPUs and GPUs are designed for different purposes, the hardware is also very different. This means that when adapting a program written to be run on a single CPU core to run on a GPU, there are many things to take into account, especially if the program has to perform well. These differences will be discussed in this section.
When a program is written to take advantage of multiple (CPU or GPU) cores, this process is called *parallelization*. When a program is adapted or written to run parts of the code on a GPU, then we say those parts of the code are *offloaded*, and the program is *accelerated*.

3.2.1. Architectural differences

As the code for this thesis is run on NVIDIA Volta and Ampere GPUs, this subsection will mostly focus on those NVIDIA GPU architectures. Most of it will translate directly to other GPU architectures, but not everything. Differences with other architectures are outside the scope of this thesis.
In general, the main differences between CPU and GPU hardware, are that a GPU not only has multiple levels of cache memory, but in general also has its own separate high bandwidth memory. Due to this separated memory, and the comparatively slow connection to the CPU system memory, this is an important difference that has to be accounted for when implementing GPU acceleration.
Another major difference between CPUs and GPUs, is that in a GPU not every core has its own cache memory. instead, multiple cores, organized in *gangs*, will share the same pool of cache memory. This means that when mapping parallelism to the cores, this sharing of memory between cores has to be accounted for ("OpenACC Programming and Best Practices Guide", 2022).
In many modern CPUs, there are multiple kinds of cores, with different levels of efficiency and performance. In NVIDIA GPUs, there are also two kinds of cores: CUDA cores, which are good at performing small numerical tasks, and Tensor cores, more specialized in certain tasks. The main GPU used for this thesis is an NVIDIA V100S card with 5120 CUDA cores and 640 Tensor cores ("NVIDIA V100 TENSOR CORE GPU data sheet", 2020). For the purposes of this thesis, only the CUDA cores are relevant.

3.2.2. Differences in performance scaling

In general, CPUs are faster when dealing with mostly sequential workloads with limited available parallelism. GPUs generally perform better when the work is highly parallel. One thing to take into account, however, is that due to the separate memory and the limited bandwidth of its connection to the main CPU memory, the amount of data needed to perform the work that is to be handled by the GPU, needs to be taken into account. If a lot of parallel work is done on a smaller problem set, a GPU will be very good at completing this task quickly. However, if the amount of work is smaller as compared to the data that is needed, then a GPU might still be slower than a CPU due to the data transfer time required to get all that data to the GPU ("OpenACC Programming and Best Practices Guide", 2022).
When deciding if it is worthwhile to implement GPU offloading to accelerate a program, it is important to keep these differences in performance scaling between CPUs and GPUs in mind. If a more precise estimate is desired of the performance gains that are attainable using GPU acceleration, this performance needs to be modeled. A general model to compare acceleration potential between different algorithms and find an estimate for the attainable speedup, will be postulated in section 3.4. This model takes into account all factors mentioned in this section.

3.3. GPU offloading implementation

There are many different ways to accelerate existing code. In this section, the different approaches that can be taken to implement this will be laid out. For this thesis, important considerations were performance, ease of implementation (due to the limited time available) with existing code, and to a lesser extend *portability*. A solution or program is considered *portable* if it is effective on many different *systems* (combinations of different hardware) or *platforms* (different software). These considerations are compared for multiple different approaches, after which a trade-off is made between them to decide which method would be used.

3.3.1. Standard parallel: do concurrent

One way to implement GPU offloading, is to specify the target GPU at compile time using compiler options, and use do concurrent to expose parallelism in loops. An advantage of this approach, is that because do concurrent is part of the Fortran standard (since 2008), every compiler can compile the resulting code and check if it has been implemented correctly.
There are however some downsides. First of all, that a compiler supports and checks do concurrent loops, does not mean it actually implements it in parallel. It also requires some minor changes to the base code, because do concurrent is implemented a bit differently as compared to standard do. Another downside of this approach is that it does not allow the programmer to regulate memory management. This means that code with a more complicated parallel structure cannot be accelerated efficiently using this method.

3.3.2. OpenACC

Alternatively, GPU acceleration can be implemented using OpenACC compiler *directives*. Those directives are placed in the original code in the form of special comments (marked by !$ACC for OpenACC) and give the compiler instructions about when it should move what data and how to map the available parallelism to the cores of the GPU.
OpenACC recognizes three different levels of parallelism: *gangs*, *workers* and *vectors*, in order from course to fine parallelism. Here each gang has its own pool of cache memory, with workers and vectors within each gang sharing the same pool. This principle is shown schematically in figure 3.1 and explained in more detail in “OpenACC Programming and Best Practices Guide”, 2022. OpenACC is

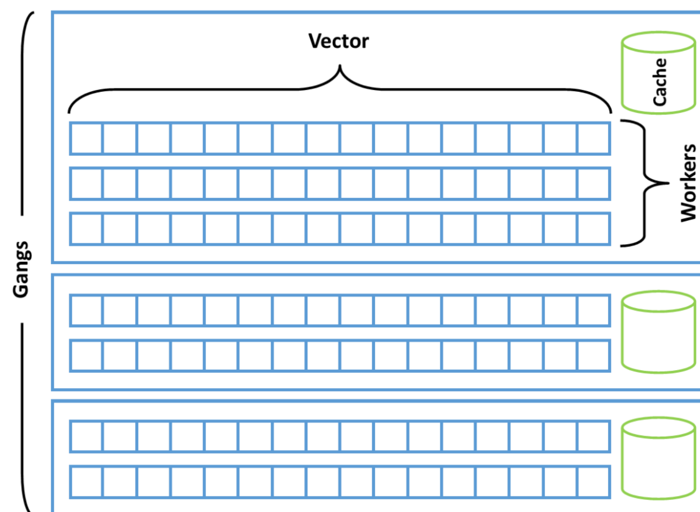


Figure 3.1: OpenACC’s three levels of parallelism.

specifically designed for GPU offloading, but it also supports multi-core CPUs. For which of the two it generates code can be determined using compiler options at compile time. Because it is specifically designed for easy GPU offloading, OpenACC is relatively easy to implement and generally yields good performance.

3.3.3. OpenMP

Another method for GPU acceleration, is through OpenMP compiler directives. In OpenMP, these directives are marked by !$OMP. OpenMP is a standard, designed for multi-core processors. It does support GPU offloading, but as it is not fully specialized, doing so is more involved as compared to OpenACC (meaning that it takes more directives to achieve the same performance and doesn't allow for as much control).

3.3.4. NVIDIA CUDA Fortran

NVIDIA CUDA Fortran is a language based on Fortran, that directly supports GPU offloading. Of the options, this option is simultaneously the hardest to implement (it requires a whole separate language and the entire code would need to be rewritten) and the one with the highest performance potential. CUDA Fortran is also designed to be optimized specifically for a given GPU architecture, making the code the least portable of the options.

3.3.5. Trade-off

In table 3.1, the different implementation methods for GPU acceleration are ranked and compared to OpenMP. 0 means that the option is about as good/bad considering the relevant criterion, +/- means it's better/worse and ++/- - means it's a lot better/worse, than OpenMP. The clear winner here is Open ACC, which is why this option was chosen for this thesis.

Method/criterion	Performance	Implementation	Portability
Std parallel	- -	++	+
OpenACC	+	++	0
OpenMP	0	0	0
CUDA Fortran	++	- -	- -

Table 3.1: Trade-off table ranking the different implementation methods that were considered for this thesis. From best of worst the meaning of the symbols is: ++, +, 0, - and - -.

3.4. Modeling GPU-acceleration potential

Amdahl's Law[1] models the parallel performance of a (semi-) parallel algorithm on a multi-core processor, based on sequential code performance, the amount of work that can be run in parallel, and the number of parallel processors (cores) available. Despite its age and the fact that the law has often been misinterpreted, the basic principle behind this law is a good starting point for predicting parallel performance for any (massively) parallel processor. When assessing the performance of GPU accelerated code, it is however necessary to also consider the time it takes to move the required data to and from the GPU, because a GPU (device) has its own separate memory, and the connection to the system (host) memory only has limited bandwidth. Another thing to take into account, is that due to the large number of parallel cores on a GPU, the number of parallel cores the code can actually take advantage of can be significantly lower than the number of cores available. This amount of available parallelism is measured by the number of available parallel threads. These two factors are explained in more detail later in this section.

3.4.1. Available parallelism and execution time

Similar to the execution time for parallel processors in Amdahl's Law as referenced before, the execution time T_p of a parallel section of code on a GPU can be estimated by, assuming that each available thread takes approximately the same amount of time to complete, the following estimator:

$$\hat{T}_p = \frac{\tilde{T}_s}{N_T} \cdot (1 + (N_T - 1)//N_C). \tag{3.1}$$

[1]The formulation of this (ancient) mathematical law from 1967 characterizing parallel computing performance and speedup compared to sequential, as well as the necessary footnotes for its interpretation, can be found in Shi, 1996. Note that the diminishing returns for most practical problems at the time predicted by this law were often misused as an argument against the development of massively parallel processors.

Here $\tilde{T}_s$ represents the sequential execution time of the section of code, adjusted for GPU hardware speed[2]. N_T is the number of parallel threads and N_C the number of cores available. $x//y$ denotes the quotient when dividing x by y, which is equal to $\frac{x}{y}$ rounded down to the next integer. In other words, calculating $x//y$ is the same as calculating x/y, but ignoring everything after the decimal point in the result. This operator is commonly used in simple computer algorithms and integer algebra. The factor in brackets calculates how many cycles the GPU needs to complete all the threads, as it can only complete N_C threads per cycle.

Equation 3.1 is derived as follows:
The parallel execution time is divided up in a number of cycles that take $\frac{\tilde{T}_s}{N_T}$ to complete (which is the time it takes a single GPU core to complete a single thread). This rests on two assumptions: all individual cores on the GPU are equal, and every individual thread takes an equal time (or work) to complete. The first assumption is, for modern higher-end GPUs, generally true, and the second assumption depends on the algorithm to be evaluated. If the amount of work between threads differs, the estimator should be used as a range, the extremes given by $\frac{\tilde{T}_s}{N_T}$ and $W\frac{\tilde{T}_s}{N_T}$, where W denotes how many times more work the longest thread is as compared to the average. Where in this range the parallel time falls depends on the scheduler and is outside the scope of this thesis.
The number of cycles is given by the term in brackets: $(1 + (N_T - 1)//N_C)$ and denotes how many times the GPU needs to run N_C cores in order to complete N_T threads, as it can only work on at most N_C threads simultaneously. This can be calculated by taking the number of full cycles $N_T//N_C$, and adding one more cycle for the remaining threads, making $1 + N_T//N_C$. This approach however creates a new problem: if $N_T = N_C$, then the GPU does not need to complete the extra cycle, because there won't be any remaining threads after the full cycles. To account for this, one is subtracted from the number of threads, making the number of cycles $(1 + (N_T - 1)//N_C)$. Multiplying the estimated time it takes to complete one cycle with the number of cycles yields equation 3.1.
Because in the real world not all threads take exactly an equal amount of time to complete, and because many different factors may affect the performance of a GPU, the estimate given by equation 3.1 is likely to be optimistic. It can however help in comparing GPU-acceleration potential of different algorithms. In a later section, these factors will be discussed and fittable form of the model is proposed to account for these factors.
If $N_T >> N_C$, i.e. when the number of parallel threads is many times higher than the number of cores, then

$$\hat{T}_p \approx \frac{\tilde{T}_s}{N_C}. \tag{3.2}$$

In most applications where GPU-acceleration is effective, this approximation is valid.[3]
On the contrary, if $N_C \geq N_T$, i.e. the number of parallel cores is higher than or equal to the number of parallel threads, then

$$\hat{T}_p = \frac{\tilde{T}_s}{N_T}. \tag{3.3}$$

For many algorithms, the number of available threads is however not a simple constant, but varies from section to section. This means that $\hat{T}$ will have to be defined as the sum over all the sections of the algorithm. In summation form, this leads to the following estimator:

$$\hat{T}_p = \tilde{T}_s \sum_{n=1}^{N} \frac{f(n)}{N_T(n)} \cdot (1 + (N_T(n) - 1)//N_C). \tag{3.4}$$

Here, N denotes the number of sections the algorithm is divided in, $f(n)$ denotes the fraction of total work the nth section represents, and $N_T(n)$ the number of threads each section could utilize.

[2]This is necessary to account for the many architectural differences between GPUs and CPUs mentioned in the previous section. $\tilde{T}_s$ can be estimated using $\tilde{T}_s \approx cT_s$, where c is some unknown constant depending on the kind of operations the algorithm consists of and what hardware is being compared.

[3]This formulation is the same as Amdahl's law for multi-core processors. This makes sense, as for multi-core processors, the number of cores is relatively small as compared to the number of cores on a GPU.

So far it was assumed that within the same region all threads take the same amount of work. If on top of that, it is assumed that even across all regions, all individual threads take the same amount of sequential work to complete, then $f(n)$ takes the form $f(n) = \frac{N_T(n)}{N_T}$. Here, $N_T = \sum_{n=1}^{N} N_T(n)$ denotes the total number of threads across all regions. Equation 3.4 takes the form

$$\hat{T}_p = \frac{\tilde{T}_s}{N_T} \sum_{n=1}^{N} 1 + (N_T(n) - 1)//N_C. \tag{3.5}$$

Note that this equation now reduces to a single fraction calculating the time a single cycle would take, and a simple sum calculating the total number of cycles needed to complete all the threads.
Because this equation doesn't give a lot of insight of how efficiently each section makes use of all the cores, the occupancy rate $O(n)$ of a few problems will be calculated to visualize this better.[4] This quantity is an estimate for which fraction of available cores can be used in section n, and defined by

$$O(n) := \frac{1}{N_C} \cdot \frac{N_T(n)}{1 + (N_T(n) - 1)//N_C}. \tag{3.6}$$

Often, like in Amdahl's law, a direct expression of the expected speedup is desired. Using the same definition of speedup as in this well known law, and plugging in equation 3.5 to estimate T_s, the speedup is estimated by:

$$speedup \approx \frac{T_s}{\hat{T}_p} = cN_T \cdot [\sum_{n=1}^{N} 1 + (N_T(n) - 1)//N_C]^{-1}. \tag{3.7}$$

Where $c = \frac{T_s}{\tilde{T}_s}$ is the constant introduced at the beginning of this section to relate the sequential CPU speed to the GPU hardware speed.

3.4.2. Data transfer time

Now that the model has been postulated to predict the time it takes for the GPU to complete a given algorithm, it is time to look into the data transfer time.
To determine the data transfer time, two things are needed: the amount of data that is moved to and from the device, and the effective data transfer rate of the connection between the host and the device. These quantities can both be estimated from the data intended to be moved to and from the GPU, and tested with a different program. In the case of modeling acceleration potential, direct calculation may be preferred.
One thing to keep in mind when determining the effective data transfer rate, is that the effective data transfer rate, in the real world, is system dependent and is often significantly lower than the specifications given by the GPU manufacturer. Two causes of this may be: bottlenecks in the connection between the CPU and the GPU (the bandwidth between two parts of a system is only as large as the smallest bandwidth of any part of the connection) and package scheduling (data is moved in packages at maximum bandwidth, but between packages, the throughput is zero) gaps.
The amount of data to be moved between the host and the device can be determined by following these steps:

1. Asses which variables need to be transferred to and from the GPU. If possible, variables should be created on the device instead of transferred.
2. Determine the size of the variable elements. For double precision real (float) variables, this is 8 bytes by default, for integers and normal (single precision) real variables, this depends on the compiler (not determined by the fortran standard), however, it can be defined manually.
3. Determine the size of the variables by multiplying the element size by the variable dimension.
4. Add up all the variable sizes to find the total amount of data that needs to be moved to and from the device.

[4]Note that this is not the same O as the $\mathcal{O}$, which is often used to denote the order of scaling of an algorithm or quantity.

After completing these steps, the resulting number can be divided by the effective data transfer rate to find the expected data transfer time. If the total completion time, including data transfer time, for the accelerated program is desired, then this number can be added to the calculated $\hat{T}_p$, found by applying equation 3.5, in order to find the total time it would take the accelerated program to complete the algorithm including data transfer time. Dividing the sequential time by this estimate will yield the expected speedup including data transfer time.

3.4.3. Miscellaneous factors

In theory, the two parts of the model above should be sufficient to predict the performance of the code with some degree of accuracy. However, in practice, there are more factors to take into account, such as kernel initialization time, natural variability of the system performance and memory allocation time. These factors are hard to predict and will be the main cause of inaccuracies of the model. Therefore, to asses performance of a given problem, it is advisable to test the hardware with an already accelerated program with a similar structure, to get a more accurate prediction of how well the newly accelerated program would perform. To account for kernel initialization times, another constant may be added to the estimator $\hat{T}_p$. Accommodating these unknowns, equation 3.5 may take the following form:

$$\hat{T}_p = \alpha \frac{T_s}{N_T} \sum_{n=1}^{N} (1 + (N_T(n) - 1)//N_C) + \beta N + \gamma N_T. \tag{3.8}$$

Here α, β and γ are fit parameters to account for unknowns such as hardware speed differences, kernel initialization times and cache memory allocation. α determines how the cycle time relates to the sequential CPU time and accounts for hardware differences. β relates the execution time with the number of parallel regions N, its main purpose is to account for kernel initialization and closing time, which means that it scales linearly with the number of parallel regions (as a kernel is started at the beginning of each region and closed at the end of said region). γ relates the execution time with the total number of threads (and with that problem size) and is there to account for cache memory allocation and access. The reason it was assumed that $T_s \propto N_T$, and not $T_s \propto N_T log(N_T)$ or just $log(N_T)$, which would be how most memory operations usually scale, is that for the accelerated program, copies of the variables are moved between memory pools (different levels of cache). The number of time a given set of variables needs to be transferred between memory locations scales with N_T. This model will be applied, calibrated and compared to the performance data in section 5.4.1.

4

Experimental method

In this chapter, the experimental method used to determine how much the calculations can be sped up using GPU offloading, is explained in detail. This includes the process of modifying the model code and optimizing for performance, as well as the method used to measure its performance.

4.1. Acceleration process

Now that the model code has been introduced, it is time to introduce the process of acceleration. The process used for this thesis, as introduced in section 3.3, is using OpenACC directives that tell the compiler what section should run in parallel on the GPU and what data to move between the host (CPU) memory and the device (GPU) memory. For more information about OpenACC and the process of GPU acceleration, please refer to "OpenACC Programming and Best Practices Guide", 2022, on openacc.org.

4.1.1. Directives and clauses

The following directives have been used in accelerating the model code:

- data construct
- parallel loop construct
- routine

A full overview of all available OpenACC directives and their clauses can be found in "OpenACC API 2.7 REFERENCE GUIDE", 2018, on openacc.org.

Data construct

The data construct is used to explicitly declare when certain objects are copied between host and device memory. The data construct surrounds a structured block of (device) code. For the data construct, the following clauses are used:

- **copyin**(*list*), on entering the structured block, objects in *list* are copied from host memory into device memory.
- **copyout**(*list*), on entering the structured block, objects in *list* are allocated in device memory. On exiting the structured block, those objects copied from device memory into host memory and deallocated from device memory (unless specified otherwise).
- **create**(*list*), on entering the structured block, objects in *list* are allocated in device memory.

Parallel loop construct
The parallel loop construct is used to explicitly declare parallelism for the next loop(s). The following clauses are used:

- **independent**, specifies that all loop iterations can be executed independently. This is usually redundant in this context.
- **gang**, distributes loop iterations over gangs (coarse grains of parallelism, as introduced in 3.3.2).
- **vector**, distributes loop iterations over vectors (fine grains of parallelism, as introduced in 3.3.2).
- **num_gangs**(*N*), explicitly declares that the number of gangs should be at most *N*.
- **vector_length**(*N*), explicitly declares that the length of each vector should be at most *list*.
- **private**(*list*), objects in *list* will be private to each thread, such that each thread has its own copy.
- **present**(*list*), explicitly declares that objects in *list* are already in device memory and should thus not be copied in when entering the loop.
- **default**(none), prevents compiler from implicitly determining data attributes for any object referenced in the loop (the use of implicit attributes is considered bad practice and often leads to errors).
- **collapse**(*N*), collapses the next *N* loops to distribute their combined iterations more efficiently over the treads.

Routine
If a routine should be executed on the device, a routine directive should be added to its definition to tell the compiler to generate device code for it. In the case of the model code, the independent loops already generate enough parallelism available to almost fully utilize the entire GPU, thus the **seq** clause was used to generate sequential code for each subroutine.

4.1.2. Data movement optimization

Moving data between the host and the device takes time. Relative to the time the device actually spends performing its tasks this is often very large, given the large amount of work a modern GPU can perform simultaneously. Therefore minimizing data movement should be top priority in the acceleration process. This can for instance be done by strategically placing data directives outside any loops and avoiding routines that implicitly initiate data transfers inside loops that run on the device. One example of this will be discussed in section 4.1.5.

4.1.3. Mapping threads across levels of parallelism

As introduced in 3.3.2, OpenACC recognizes three separate levels of parallelism. As the three levels are handled very differently by the GPU, it is often worth it to test a few different levels of parallelism and different vector lengths or number of gangs to test which configuration yields the best results. Keep in mind that specifying vector length may harm the portability of the code, as different GPUs will have slightly different optimal vector lengths, so if the code needs to run on many different GPUs, it may be best to let the compiler determine the vector length automatically. (“OpenACC Programming and Best Practices Guide”, 2022)
One thing to also keep in mind when testing different configurations of gangs, workers and vectors, is that because the three levels of parallelism access memory in different ways (remember for example that only gangs have their own cache). This means that whenever this structure is changed, the code needs to be tested again and compared to the original code to check for correctness. During the experiments, it was for example found that most loops performed best with a gang-vector structure, and that distributing threads among multiple workers often led to incorrect results.

4.1.4. Exposing more parallelism

If the number of parallel threads available isn't already large enough to fully utilize the GPU, more parallelism can be exposed by restructuring loops and adapting subroutines. However, as the original code already has way more parallelism available than the GPU can take advantage of, potential gains should be minimal. This step in the acceleration process is also relatively time-consuming when applied to existing code as it often requires large portions of the code to be completely restructured. One example where more parallelism was added, is the plane sweep algorithm, as in this case the first and last few planes don't have enough parallelism to fully take advantage of the many cores of a GPU. The results of this will be analyzed in section 5.1.

4.1.5. Further optimizations

Further improvements in performance can be achieved by performing lower level optimization on the base code apart from the OpenACC directives. Among those optimizations are eliminating assumed shape arrays from subroutines and functions in accelerated code, as well as the substitution of more specialized subroutines and functions in place of fortran intrinsic functions.
Assumed shape arrays require the program to fetch certain metadata from the host memory about the exact shape of input arrays and need to communicate metadata about the shape of output arrays back to the host, whenever a subroutine that uses them is called from accelerated code. This causes unnecessary data transfers which slow down the accelerated code and may cause disappointing returns. This means that eliminating them from any subroutine that is called from the device has the potential to significantly improve performance. (Appelhans, 2023)
The fortran standard contains many intrinsic functions for executing basic operations, such as matrix-matrix products and dot products. These functions are useful for easily building portable code that doesn't require the programmer to import or write any new functions or subroutines. These intrinsics are however often built with cpu programming in mind and are often relatively flexible in their inputs. These intrinsics may perform very or reasonably well in certain scenario's, such as cases where large matrices need to be multiplied from a sequential to coursely grained parallel algorithm, but may actually be significantly slower or even completely break the program when used in different circumstances.
One relevant example of this issue is the matmul intrinsic function. When used inside a parallel environment, the matmul intrinsic significantly slowed down the program (for further details see 5.1 and 5.2), and even broke the code completely in nvfortran 22.3, due to the generation of new undeclared arrays. Replacing it with a dedicated sequential algorithm with explicitly declared arrays solved these issues.

4.1.6. data representation

One factor that greatly impacts data transfer and computation time is the precision at which data is represented. One paper focusing on the impact of data representation on GPU accelerated computational fluid dynamics (cfd), for example, reports cases with performance gains of up to 2.9x without a significant loss of accuracy. (Witherden & Jameson, 2020)
Until this point, all the calculations for this thesis have been computed at double precision (each floating point number is represented by 8 bytes, which allows for 16 digits of precision), which means that the results may be very precise, but this does come at an increased computational cost. The FEM plane sweep algorithm implementation is also partially tested at single precision (4 bytes instead of 8, yielding only 7 digits. This is implemented by substituting double precision variables with real(4) ones in the source code.) to measure the impact precision has on computing times, as well as to find an estimate to how this impacts the accuracy of the result.

4.2. Performance measurements

In any experiment, the way things are measured has an impact on the results and is crucial for the conclusion of the research. That's why this section will focus on the conditions in which the experiments are conducted, as well as the methods in which the results in the next chapter have been generated.

4.2.1. Hardware

Both GPU and single core performance have been measured on the GPU nodes of the Delft Blue supercomputer. Debugging and optimization have for the biggest part been performed on the Apollo machine at the Faculty of Applied Sciences in Delft. This choice was made because this machine, contrary to Delft Blue, allows for direct testing with no queues. This however does mean that results on this machine vary significantly between different runs and from day to day depending on how much the computational resources are used by other researchers. Additionally, the Apollo machine also has the NVTX toolbox installed, which contains the profiling tool Nsight, as described later in this section. The multi core performance has been measured on a separate compute node on the Delft Blue supercomputer. Specification of the hardware on both nodes on Delft blue supercomputer, as well as the the Apollo machine are provided in table 4.1. ("DelftBlue Hardware", 2023)

Specification	GPU node Delft Blue	Compute node Delft Blue	Apollo
CPU	2x AMD EPYC 7402 @ 2.80 GHz	2X Intel XEON E5-6248R @ 3.0 GHz	2x Intel Xeon Silver 4214R @ 2.40 GHz
Number of cores (total)	48	48	24
Memory (GB)	256	192	?
GPU	4x NVIDIA Tesla V100S, 32 GB	N/A	3x NVIDIA Ampere A40, 48 GB

Table 4.1: Hardware available on the different machines. On Delft Blue memory is specified for each run to not cause unnecessarily long wait times and allow enough space for other users. On Apollo all memory is always available, but has to be shared among processes.

4.2.2. Compilers

To test the performance portability of the code, and to provide a more representative comparison for performance on the CPU, multiple different compilers are tested. A general overview of the compilers that have been used is provided in table 4.2 below.

Distributor	Compiler	Version number	Accelerator support
Intel	ifx	2022.2.0	OpenMP
GNU	Gfortran	11.2.0	OpenACC* and OpenMP
NVIDIA	nvfortran	22.3 (on Delft Blue) and 22.5 (on Apollo)	OpenACC and OpenMP

Table 4.2: Compilers used during the experiments. *According to the documentation, OpenACC support is still in the initial phase in Gfortran 11.2. ("The GNU Fortran Compiler 11.2.0", 2021)

4.2.3. Compiler options

When compiling the programs for performance testing, the compiler options that are used have the potential to greatly impact the performance of the executables. The optimal compiler options depend on both the compiler itself and the hardware available. In this subsection, the compiler options used are explained in detail for each tested scenario.

Compiling for single core CPU performance comparison

To calculate the speedup achieved by GPU acceleration, a comparison is needed. For this comparison, the same source code is compiled and tested on a single CPU core for multiple compilers as well. The compiler options that are used for this purpose are shown per compiler in table 4.3 and explained in table 4.7 at the end of this section.

Compiler	Options
Intel ifx	-O -inline
GNU Gfortran	-O -Winline -ffree-line-length-none
NVIDIA nvfortran	-O -Minline

Table 4.3: Compiler commands used for single core performance testing.

Compiling for GPU on Delft Blue
To calculate the speedup achieved by GPU acceleration, the GPU performance of course needs to be measured. The compiler options that are used for this purpose are shown per compiler in table 4.4 and explained in table 4.7 at the end of this section.

Compiler	Options
Intel ifx	<OpenACC not supported>
GNU Gfortran	-O -Winline -ffree-line-length-none -fopenacc
NVIDIA nvfortran	-O -Minline -acc=gpu -gpu=cc70,cuda11.6 -gpu:pinned

Table 4.4: Compiler commands used for GPU performance testing on the Delft Blue supercomputer.

Compiling for GPU on Apollo
For rough optimization, correctness checking and general bugfixing, the long wait times (often on the scale of hours) of the queue at Delft Blue makes it not very suitable for quick iterations. Therefore these processes were often performed on a separate machine called Apollo. As the hardware is slightly different (see table 4.1) on this machine, the compiler options for nvfortran are slightly different. The compiler options that are used for this purpose are shown per compiler in table 4.5 and explained in table 4.7 at the end of this section. The directives used for CPU comparison are the same as earlier.

Compiler	Options
Intel ifx	<OpenACC not supported>
GNU Gfortran	-O -Winline -ffree-line-length-none -fopenacc
NVIDIA nvfortran	-O -Minline -acc=gpu -gpu=cc86 -gpu:pinned

Table 4.5: Compiler commands used for GPU performance testing on Apollo.

Compiling for multicore CPU comparison
Because GPU's are much more expensive than a single CPU core, as a GPU costs at least as much as a full multi-core CPU, it is only fair to see if GPU performance is that much better as compared to a multi core CPU. In order to test this, a small number of tests have been performed using the OpenMP directives in the original model code on the Intel ifx compiler, as well as the OpenACC directives on the NVIDIA nvfortran compiler, with a multicore cpu as target. The compiler options that are used for this purpose are shown per compiler in table 4.6 and explained in table 4.7 at the end of this section. The directives used for CPU comparison are the same as earlier.

Compiler	Options
Intel ifx (MP)	-O -inline -qopenmp -qmkl
NVIDIA nvfortran (ACC)	-O -Minline -ta=multicore

Table 4.6: Compiler commands used for CPU multi-core performance testing on the Delft Blue supercomputer.

Option	Description
-O	Tells the compiler to optimize the executable for speed as much as possible, at the cost of slightly longer compile time and risk of correctness. By default compilers are very conservative in this, sacrificing performance for a slightly lower chance of mistakes. Impact on correctness was tested and no mistakes were found as a result of this option.
-inline/Winline/Minline	Tells the compiler to insert functions and short subroutines right where they are called. This slightly improves performance as it eliminates some lookup overhead.
-ffree-line-length-none	This command is necessary to compile longer lines of code in Gfortran, as it has a hard line length limit of 132 characters by default (which some OpenACC statements easily exceed). This is common in legacy compilers, but for modern compilers this portability feature is redundant.
-fopenacc	Tells Gfortran to interpret OpenACC directives.
-acc=gpu	Tels nvfortran to write accelerator code for a GPU.
-gpu=cc<xy>,cuda<#>	Tells nvfortran what compute capability x.y (generation) gpu it is compiling for. Also tells it what version of cuda it should use (only necessary on Delft Blue).
-gpu:pinned	Tells nvfortran that it should use pinned memory by default. This may improve memory transfer speed, but comes at a cost of longer variable initiation time. Mileage varies a lot, so code is also always tested without this option enabled.
-qopenmp	Tells ifx to interpret OpenMP directives.
-qmkl	Tels ifx to link the lapack module, which is used in the original code to solve the 4x4 system.
-ta=multicore	Tells nvfortran to use a multicore cpu as target for acceleration.

Table 4.7: A general overview of all relevant compiler options.

4.2.4. Code segments

For this thesis, the performance of the main loop over all elements of the grid and the time it takes to transfer all necessary data to and from the GPU are of interest, whilst the rest of the code is only there to support it. This means that the code has to be divided up into segments, over which the time difference can be measured. The first segment, copy in, starts before the first data is moved to the GPU and ends before the main loop starts. It measures how long it takes to copy all necessary data to the GPU. The next segment, ”sweep”[1], contains the main loop over all the elements and measures how long it takes the GPU to complete the task. The final segment, copy out, starts directly after that and measures how long it takes to copy all info off of the GPU back to system memory. The first and final segments are added together to get the final data transfer time, that will be reported next to the sweep times in the next chapter.

[1]For the purposes of this thesis, the main loop is referred to as ”sweep” in both the plane sweep algorithm and the matvecs algorithm.

4.2.5. System clock

Now that the segments of interest have been defined, it is time to define the measuring method. The time the code spends in a given segment is measured by calling the ”system clock” intrinsic on the device. It saves the current system time to a variable, which allows the program to simply calculate the time difference between the two calls responding to both variables. The accuracy of this intrinsic depends on the system clock rate, which is determined by the typing of the rate variable supplied to the count_rate parameter on initialization. This dependency varies wildly between different compilers however. In the model code, a default integer (4 bytes) is used for this, which means that for example the Intel ifx compiler has a resolution of 100μs whilst NVIDIA nvfortran counts with a resolution of 1μs. Given the order of magnitude of the time the code spends in a given segment (shortest measured time in ifx was in the order of hundreds of milliseconds), this however does not cause any problems. To the reliability of the results. In the lines below, an example is given of how this function is initiated and how it returns the run time of the enclosed code segment in seconds.

```
integer :: cr,cm,rate,time0,time1

CALL system_clock(count_rate=cr)
  CALL system_clock(count_max=cm)
  rate = REAL(cr)

call system_clock(time0)
```

!!!Code segment!!!

```
call system_clock(time1)
print *,'run time',real(stime1-stime0)/rate
```

4.2.6. Profiling tools

During the debugging and optimization process, a profiling tool called NVIDIA Nsight is used, to verify that the code is executed as expected and to verify issues like long kernel initiation times and unexpected data transfers. It also allows the user to trace specific regions of code using NVTX markers, to see how long the program spends in those regions, and what kind of behavior correspond to what region. Using a profiling tool however creates significant performance overhead to trace all processes that are happening, which means that it cannot be used to accurately measure performance. In figure 4.1, an example is shown of a GPU profile obtained using this profiler. The bottom track shows NVTX markers, which show that the GPU is currently carrying out instructions corresponding to the code region ”plane jump”. In the example, extra data transfers in between parallel regions (caused by a subroutine using assumed shape arrays) are visible (purple and green lines in the memory track). Adapting the code to not need those extra data transfers significantly improved code performance.

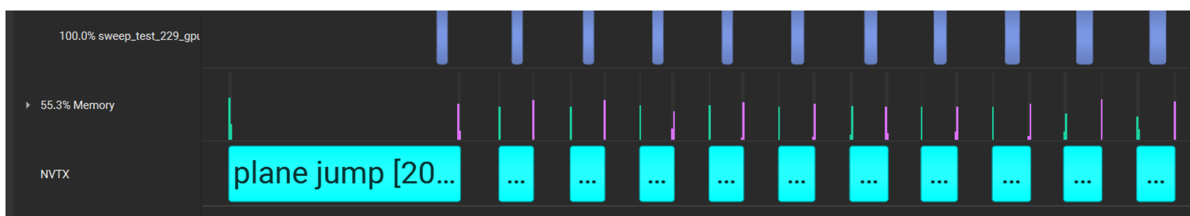


Figure 4.1: Example of a GPU code profile.

5

Results and discussion

In this chapter, the performance of the accelerated code on the GPU is compared to the sequential code running on a CPU for multiple different problem sizes, and the general mathematical model as postulated in section 3.4 is compared to the measured performance of GPU accelerated code. All results in this chapter were tested using double precision data representation, unless specifically stated otherwise. The results used to generate the graphs in this chapter are shown in table form in appendix C. The impact of some different optimizations mentioned in the previous chapter are also included.

5.1. Plane sweep

For the plane sweep algorithm, the problem size is determined by the number of elements in the spatial discretization grid and the number of sweep directions considered. The problem size (P), i.e. the number of computations required to solve a case, scales with the following relation:

$$P \propto N_O \cdot N_E, \tag{5.1}$$

where N_O denotes the number of orientations of $\hat{\Omega}$, and N_E denotes the number of spatial discretization elements in the grid. Note that for a cubic grid of axis length ℓ, N_E scales with ℓ^3, if element volume remains constant.

For sequential code, computation time is usually directly proportional to problem size. For highly parallel code, this is however not always the case. In figure 5.1, the computation time of the GPU accelerated code, with and without data transfers, and the sequential code are visualized and compared for different parameters N_O and N_E. In figure 5.2, the speedup for those different parameters is visualized and compared with the relative problem size (relative to the $1x128^3$ case). In the first two figures, problem size is also shown, but scaled with the sequential time of the smallest problem for easier comparison. For case naming, the following naming convention is used: $N_O x\ell^3$. This convention is chosen because it also doubles as a representation of the relative problem size: the problem size of $2x256^3$, for example, is twice as large as that of 256^3 and eight times (2^3) as large as $2x128^3$. As will become clear from the graph, problem size alone is not sufficient to accurately predict how much performance can be gained using GPU-acceleration. However, larger problems are more likely to benefit as compared to smaller ones.

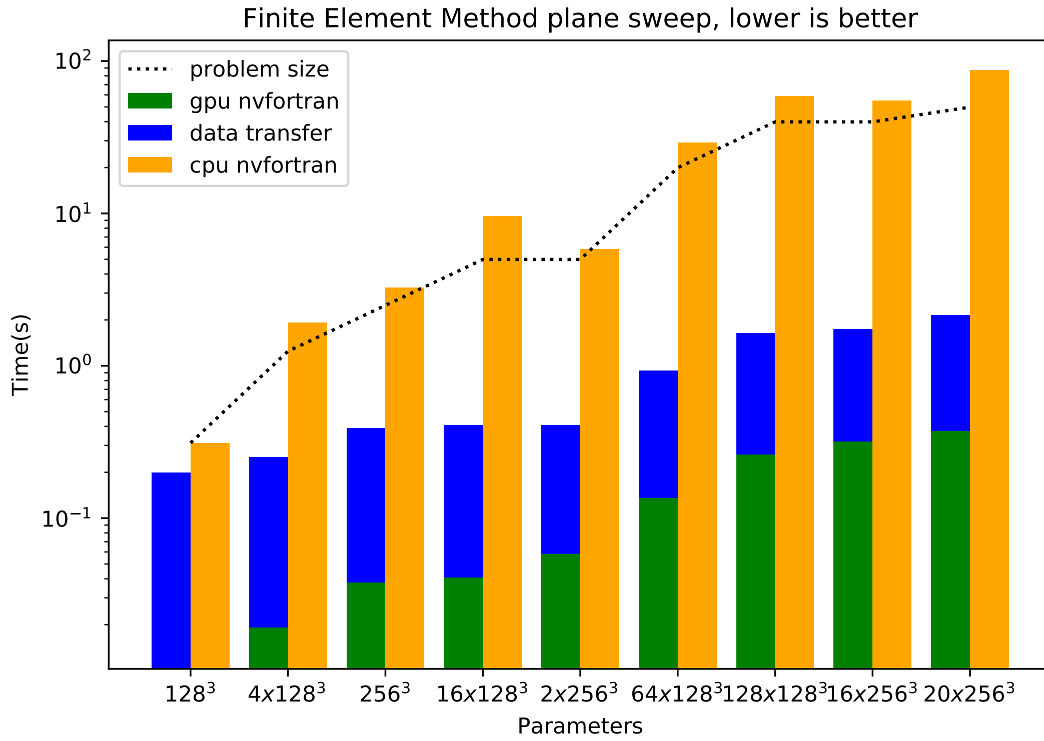


Figure 5.1: Sweep and data transfer time for different problem size parameters N_O and N_E, for code run on the GPU and sequentially on the CPU. The code corresponding to this graph was compiled using nvfortran 22.3, as described in section 4.2.3. For axb on the x-axis, a represents the number of discrete sweep directions and b the number of elements. The dotted line represents the relative problem size as calibrated to the 128^3 case.

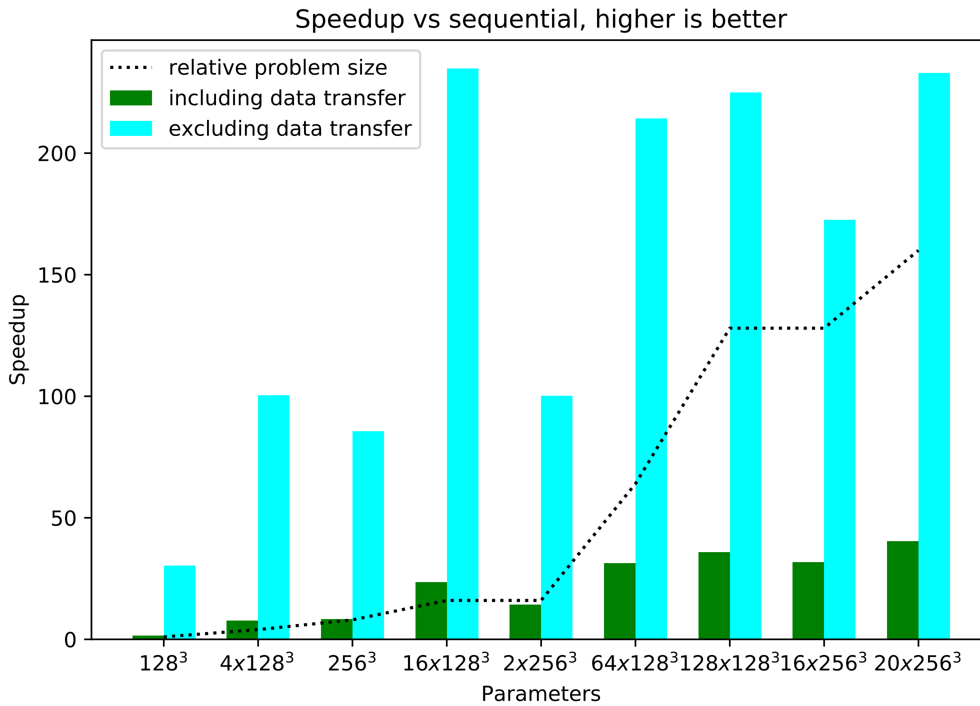


Figure 5.2: Speedup of the GPU accelerated code as compared to sequential CPU code, with and without data transfer time taken into account. Larger problems seem to allow for higher speedup.

5.1.1. Pinned memory

As stated in section 4.2.3, instructing the compiler to define variables in pinned memory may improve performance. As becomes apparent from figure 5.3, defining variables as pinned did however not lead to consistent performance gains for the plane sweep algorithm.

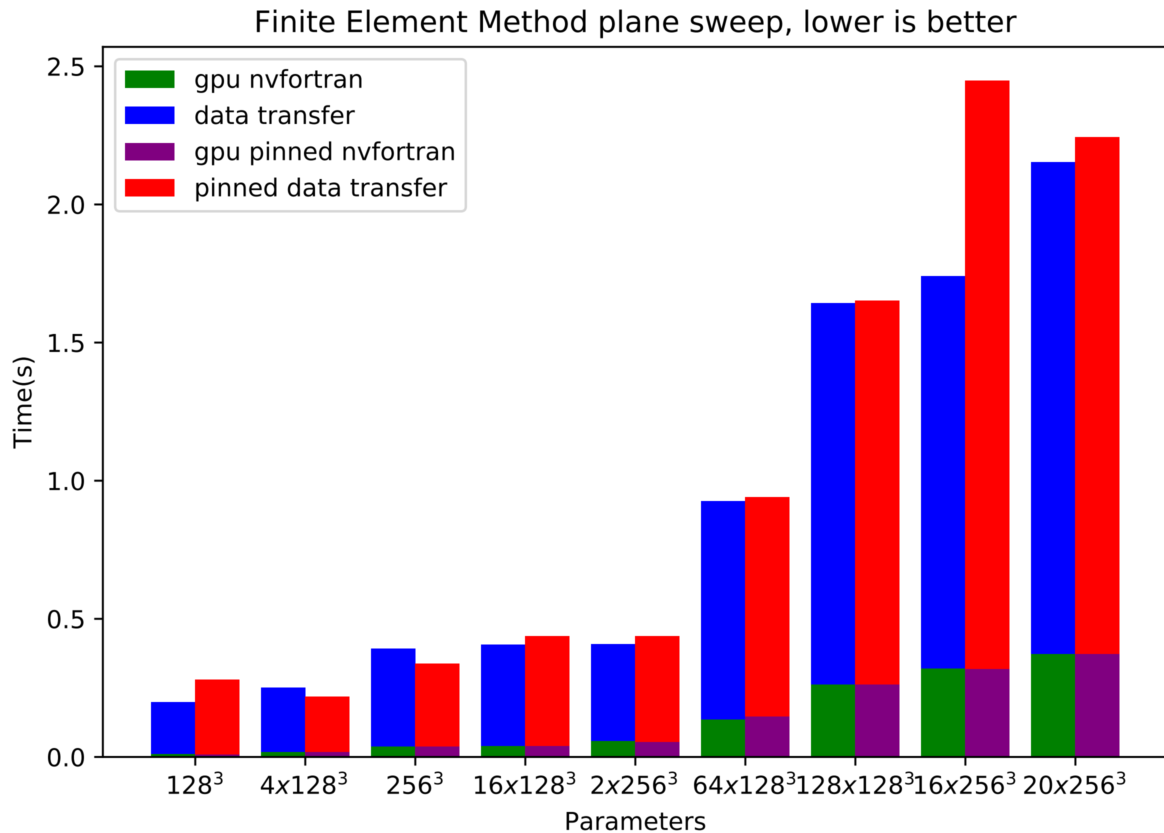


Figure 5.3: Sweep and data transfer time for various problem sizes, with and without specifying pinned memory.

5.1.2. Multicore

As stated in section 4.2.3, the model code was also tested on a multi-core CPU. The results were however a bit disappointing for the plane sweeping algorithm; the original OpenMP implementation for a 256^3 element grid performed worse on multiple cores as compared to sequential code: roughly 8 seconds wall-clock time for every iteration of the parallel code, instead of just 5 for the sequential algorithm, as tested with 16, 24, 32 and 48 cores.
The OpenACC implementation of size $4x128^3$ has also been tested on a multi-core as described in section 4.2.3 and was only 20% faster as compared to the sequential code. The result generated by this multi-core implementation however differs by multiple percentage points from the sequential implementation, as will be briefly discussed in a later section. As multi-core performance is not the focus point of this thesis, the cause of these disappointing results on a multi-core CPU has not been further investigated.

5.1.3. Data representation

In section 4.1.6, the impact that data representation may have on the code performance is mentioned. The loss of accuracy as a result of using single precision instead of double precision will be discussed in a later section. The performance of the $128x128^3$ with single precision data representation has been measured for both sequential CPU code and GPU accelerated code and is compared to the relative performance of its double precision counterparts in figure 5.4. As becomes apparent from the two graphs, the data transfer time sees the most significant reduction when single precision is used as opposed to double precision.

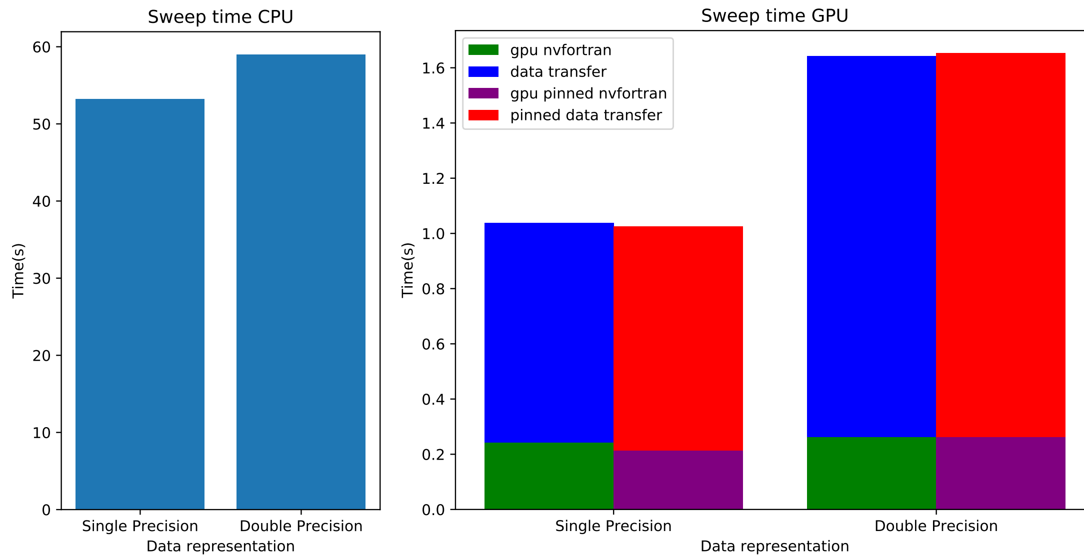


Figure 5.4: On the left, sweep time for the sequential CPU code is shown for single precision and double precision data representation. On the right, the data transfer time and sweep time of both representations are visualized for both pinned and non-pinned GPU code.

5.1.4. Gfortran and ifx

As compiler choice has a very significant impact on the performance of the resulting executable, multiple compilers are tested for generating the sequential CPU code. For the GPU code, nvfortran was the only compiler available on Delft Blue that could compile instructions for the GPU that was used for this thesis. As both Gfortran and ifx have a limited array size, causing fatal truncation errors when compiling code for larger problem sizes, only the smaller problems were tested with those two compilers and nvfortran was chosen as the reference compiler for sequential comparisons instead. The sequential performance of those three compilers is compared in figure 5.5.

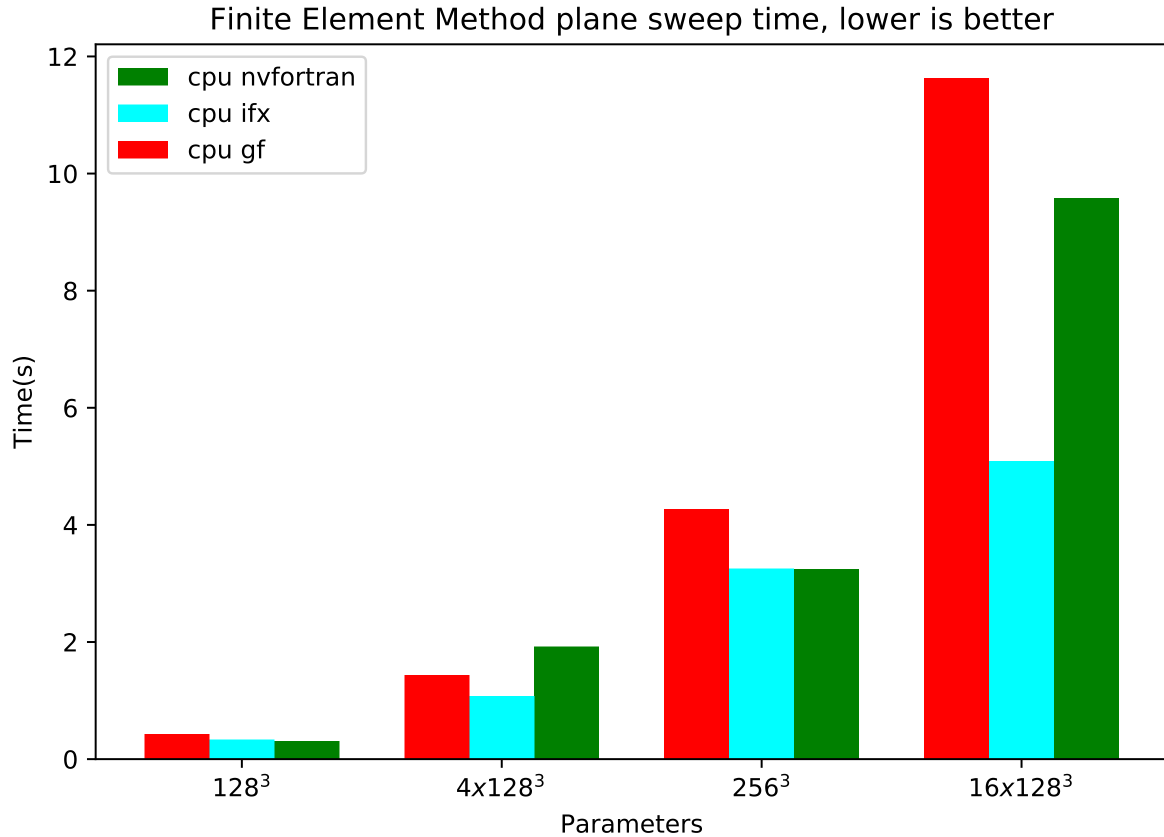


Figure 5.5: Sweep time of sequential code compiled by nvfortran, ifx and Gfortran (gf) compared. Due to issues with array representation for larger problems in both ifx and Gfortran, only the smaller problems are compared in this figure.

5.1.5. Correctness

For each test of the accelerated code, the norm of the resulting $\vec{\phi}$ vector[1] was calculated (and summed for all discrete directions) and compared in order to check the correctness of the result. It was found that for each set of parameters, all sequential and GPU accelerated code compiled using nvfortran and Gfortran was always in exact agreement, where ifx deviated from the other two by a very small (roughly $1 \cdot 10^{-9}\%$) margin. The latter was also in exact agreement with the original model code, as compiled with the ifx compiler.

Results generated using single precision deviated by roughly 0.1% from double precision results and CPU multi-core (using nvfortran with OpenACC) results deviated by roughly 3% from sequential code. Results generated using ifx with OpenMP were in agreement with results generated using ifx without OpenMP, but the performance of the parallel code using this method was worse as compared to the sequential code, as mentioned in a previous subsection.

[1]This $\vec{\phi}$ vector is not a vector in the ordinary sense. This vector contains four numbers for each element in the grid, corresponding to the four first order polynomial constants modeling flux as a linear function through the element cell.

5.2. Matvec

Similarly to the plane sweep algorithm, the performance of the GPU-accelerated FEM implementation of the matvec algorithm has also been tested for multiple problem sizes. Because of the large amount of available parallelism already available in the matvec algorithm, however, this algorithm has only been tested with a single flow direction. The sweep and data transfer time of the GPU-accelerated code is, like for the plane-sweep algorithm, compared to the sequential CPU code in figure 5.6.
For problem size 256^3, the matvec algorithm was also run on a multi-core CPU, using nvfortran with OpenACC. In case of the matvec algorithm, the multi-core CPU generates accurate results and has good performance. However, if the data is already present at the GPU for a different computation, it is quicker to execute the matvec at the GPU before transferring data back to the host. The multi-core performance is included in this figure for comparison purposes.

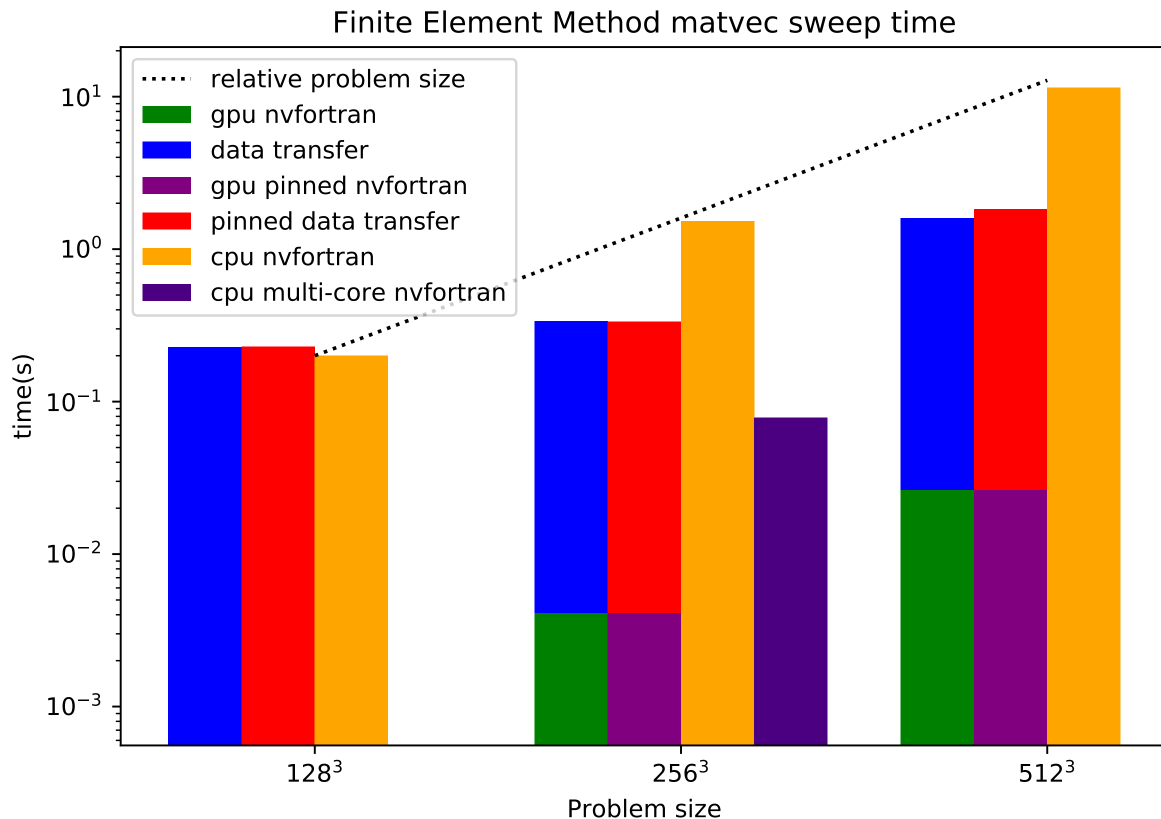


Figure 5.6: Performance of the matvec algorithm, as compiled using nvfortran. The relative problem size is also plotted (as calibrated to the smallest case), to more clearly represent the scaling between the cases.

Just like with the plane sweep algorithm, the data transfer time is orders of magnitude longer as compared to the time it takes the GPU to execute the algorithm itself. Also of note is the multi core completion time, which is an order of magnitude better as compared to sequential performance and significantly better as compared to GPU performance if memory transfer is included. If the data is already on the GPU, or needs to be on the GPU for a different algorithm, the GPU is significantly faster as compared to even the mullti-core processor. Again, declaring pinned memory did not yield any performance benefits.
The measured speedup of the GPU-accelerated code as compared to sequential, excluding data transfer, is 360x, 370x, and 440x for problems of sizes 128^3, 256^3, and 512^3 respectively. The same metric including data transfer would be 0.88x (including data transfer time, the smallest problem is slower on the GPU, as compared to the CPU), 4.5x, and 7.2x.
These different proportions for the matvec algorithm as compared to the sweep test algorithm can be explained by comparing the parallel structure, as well as the amount of work versus the amount of data needed; the matvec algorithm is fully parallel (explaining the large speedup excluding data transfer),

but also requires more data to be transferred as compared to the amount of work done by the algorithm, explaining the large impact data transfer has on the achieved speedup. The achieved CPU multi core (48 cores) speedup was 19.4x as compared to sequential for problem size 256^3.
Like with the plane sweep, the compiler choice also has an impact on the performance. For sequential execution of the smallest two problem sizes, the sweep time is compared in figure 5.7. Due to some limitations with the ifx compiler and the gfortran compiler, in defining larger arrays, only the smaller two problems were tested.

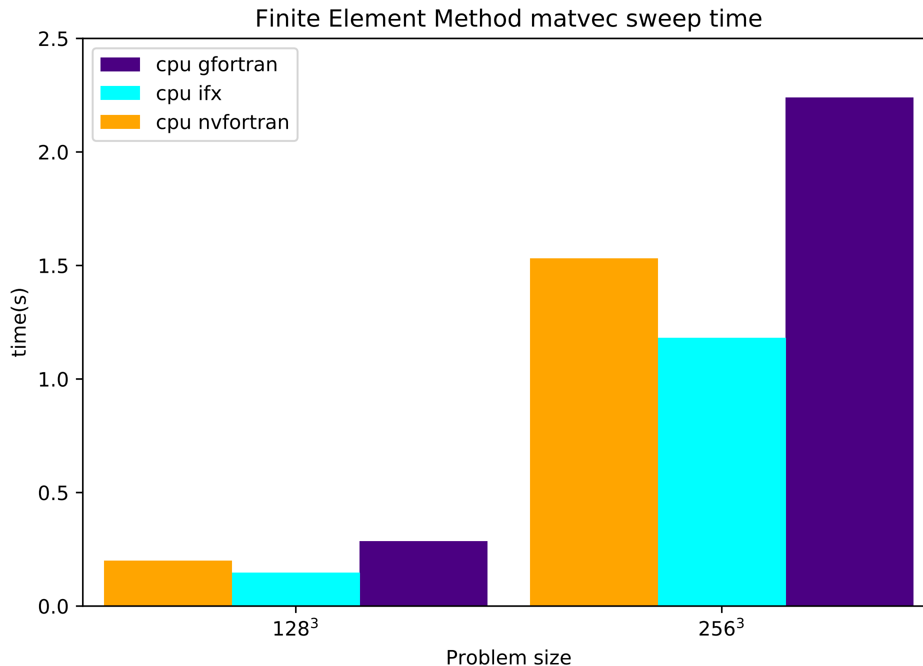


Figure 5.7: Which compiler is used has a significant impact on the results. Due to the array size limitations in ifx and Gfortran, mentioned when discussing the plane sweep algorithm, only the first two sizes are shown.

5.3. General implementation considerations

During this project, a newer version of nvfortran became available (version 23.5). Preliminary testing with this newer version has shown no changes in performance for the sequential CPU code and GPU-accelerated code, with a slight reduction in data transfer time. However, more research is needed to fully rule out any effects this new version may have on the results.
For some larger problem sizes, the plane sweep algorithm requires a lot of data to be stored on the GPU. The largest problem that was tested, which was case $20x256^3$ for example, had a $\vec{\phi}$ vector which took up over 10 GB of memory, with on top of that many more large variables that needed to be stored. An even larger problem that was tried, case $64x256^3$, would require over 34 GB of data to be stored just for the $\vec{\phi}$ vector, which is more then most GPUs have available. When implementing this GPU-accelerated algorithm in the real world, measures need to be taken to reduce the amount of memory required on the GPU, if it should be able to handle larger problems.
Significant performance gains can be made if both algorithms are run back-to back, without first transferring the data back to the host and to the device again. This could drastically decrease total computation time, as most of the data transfer time for the plane sweep algorithm is spent at the end, whilst most of the data transfer time for the matvec algorithm is spent at the beginning. For large problems, the amount of ”intermediate” data transfer could potentially be up to 90%. To achieve this however, all the steps in between should also be adapted to run on the GPU. Another hurdle could be memory management, as both algorithms require large arrays to be stored on the GPU, which do not fully overlap. All in all, more research is needed before the feasibility of this approach can be determined.
During the project, all code was compiled on a separate node with slightly different hardware, because this node was directly accessible, making it easier to monitor the compilation process. In a later phase

in the project, the difference in performance was assessed between compiling on a separate node, and compiling right before runtime on the destination node (which means that the code is run on the same hardware as the hardware it is compiled on). It was found that there was no difference in compute performance (i.e. the sweep time was the same), however, the data transfer times were found to be roughly 20% shorter as compared to compiling on the separate node. This means that the speedup including data transfer, as documented earlier in this chapter, may be more on the pessimistic side, with real world performance likely better than reported. One explanation for this behavior, is that cross-compiling (i.e. compiling on different hardware than the program is run on) may cause the compiler to make wrong assumptions about the system the code will run on, generating code that may run slower, or as in this case, move data less efficiently.

5.4. Applying the general performance model

In section 3.4, a general model to estimate the expected speedup of GPU-accelerated is postulated. In this section, this model will be applied to the test cases and compared to the test results.

5.4.1. Execution time

The amount of available parallelism represents the biggest difference between both algorithms that were tested. The matvec algorithm, for instance, is fully parallel. This means that estimating the execution time excluding data transfers can be done by using equation 3.2, as all parallelism is on a single section and $N_T >> N_C$ holds (the smallest test case already has $128^3 = 2097152$ threads available). This equation reads

$$T_p \approx c\frac{T_s}{N_C}, \tag{5.2}$$

where c is some fittable constant depending on the type of operations the algorithm contains, and the hardware that is compared. In figure 5.8, the model has been fitted to the performance data of the algorithm on the GPU. In blue, extra measurements have been included to validate the predictions made by the model.

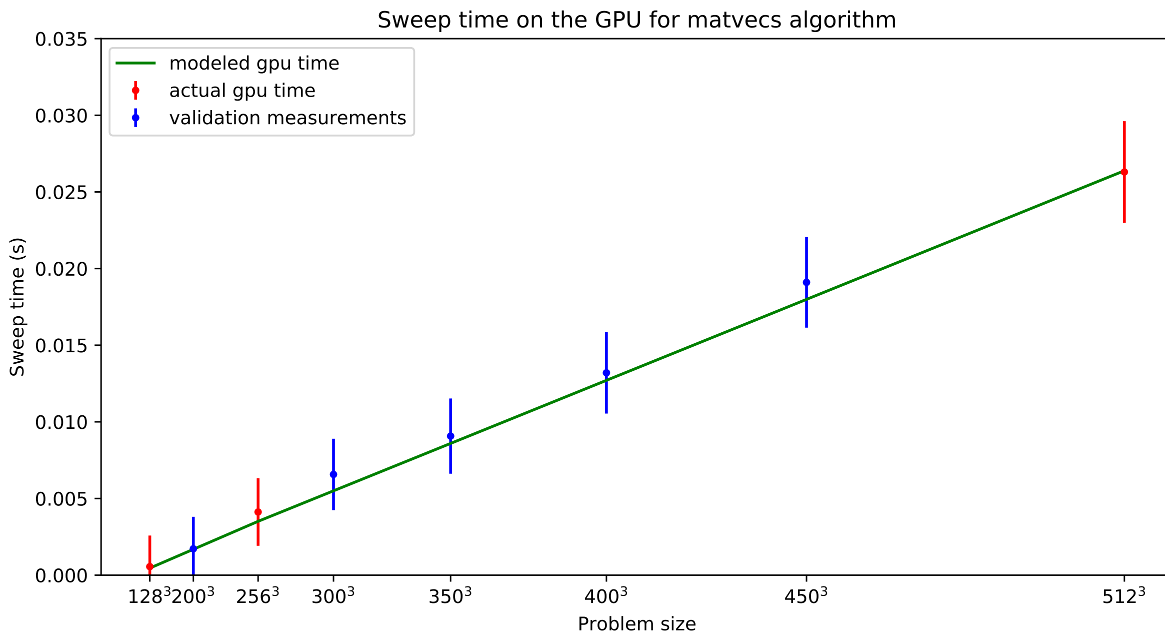


Figure 5.8: GPU execution time of the matvec algorithm of three different problem sizes, fitted to GPU performance data.

For this algorithm, c was found to be roughly equal to 12. The error bars of both the initial data, as the validation data set, were set to 5%, with an offset of 2 ms, to account for variations encountered when measuring the performance.

The available parallelism in the plane sweep algorithm, contrary to the matvec, varies as the algorithm progresses from plane to plane. This means that the estimated execution time for $T_{p,planesweep}$ has to be estimated using equation 3.8. Therefore the following estimator is used:

$$\hat{T}_{p,planesweep} = \frac{\alpha T_s}{N_T} \sum_{n=1}^{N_{planes}} 1 + (N_T(n) - 1)//N_C + \beta N + \gamma N_T, \tag{5.3}$$

where N_{planes} denotes the total number of planes, T_s denotes the completion time of the sequential algorithm on a CPU, N_C denotes the number of available cores, and n denotes the current plane number in the summation. Like in the original model, N_T denotes the total number of threads throughout the algorithm, whilst $N_T(n)$ denotes the number of threads available in a given section. In figures 5.9 and 5.10, this model (green) is fitted to the GPU performance data (red).
The error bars are determined by the natural variability of the measured performance (which is estimated at 10%) and a constant measuring error of 5 ms. The modeled execution time is shown in green. The optimal values for the fitting constants, as fitted to both the 128^3 and the 256^3 problem sets were found to be: 0.250, $3.28 \cdot 10^{-5}$ and $2.11 \cdot 10^{-5}$ for α, β and γ respectively.

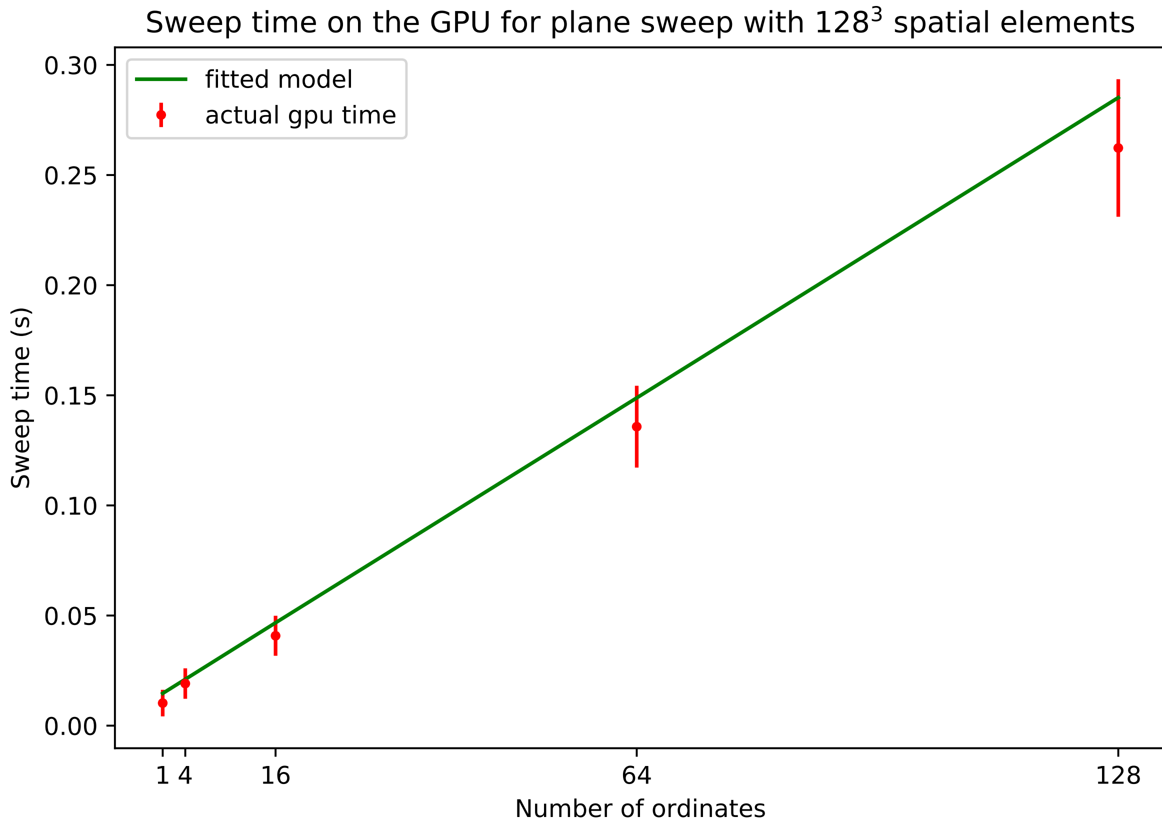


Figure 5.9: GPU execution time of the plane sweep algorithm with 128^3 elements, tested with different numbers of ordinates.

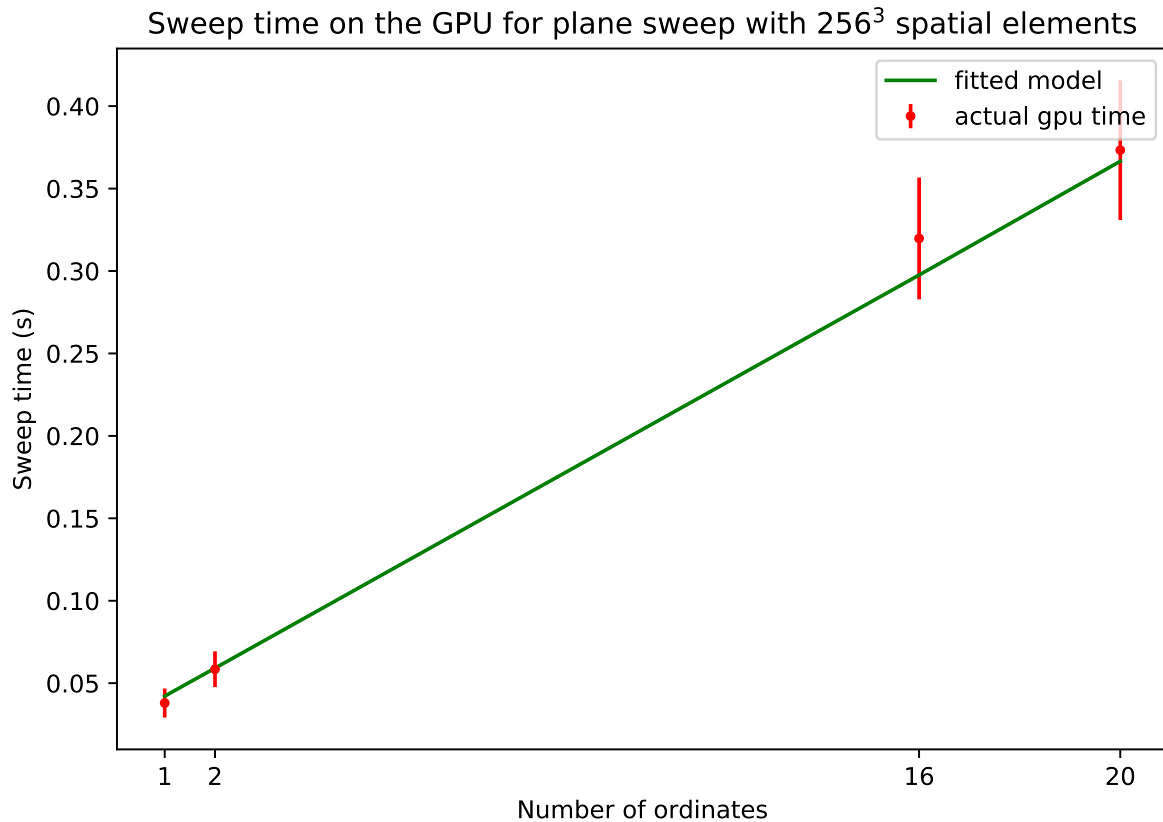


Figure 5.10: GPU execution time of the plane sweep algorithm with 256^3 elements, tested with different numbers of ordinates.

From the graphs it becomes apparent that the model tends to overestimate the time it takes to complete problems with 128^3 elements, whilst underestimating the problems with 256^3 elements. However, the model still correctly describes the results within the margin of error. More research is needed to verify the accuracy of this model.

To understand why this line is relatively flat (computational time does for example not increase by fourfold between 1 and 4 ordinates, and is also not eight times larger between the 128^3 and 256^3 element grids), it is helpful to analyze the available parallelism, as it gives insight in how efficiently the GPU resources are used. For this purpose, 5.11 shows the available parallelism for each plane in the three smallest cases of the plane sweep algorithm, and how it maps onto the cores of the Nvidia A100 GPU, which is the GPU used to perform the measurements during this research. The number of (CUDA) cores on such a GPU is 5120, according to the spec sheet provided by Nvidia.

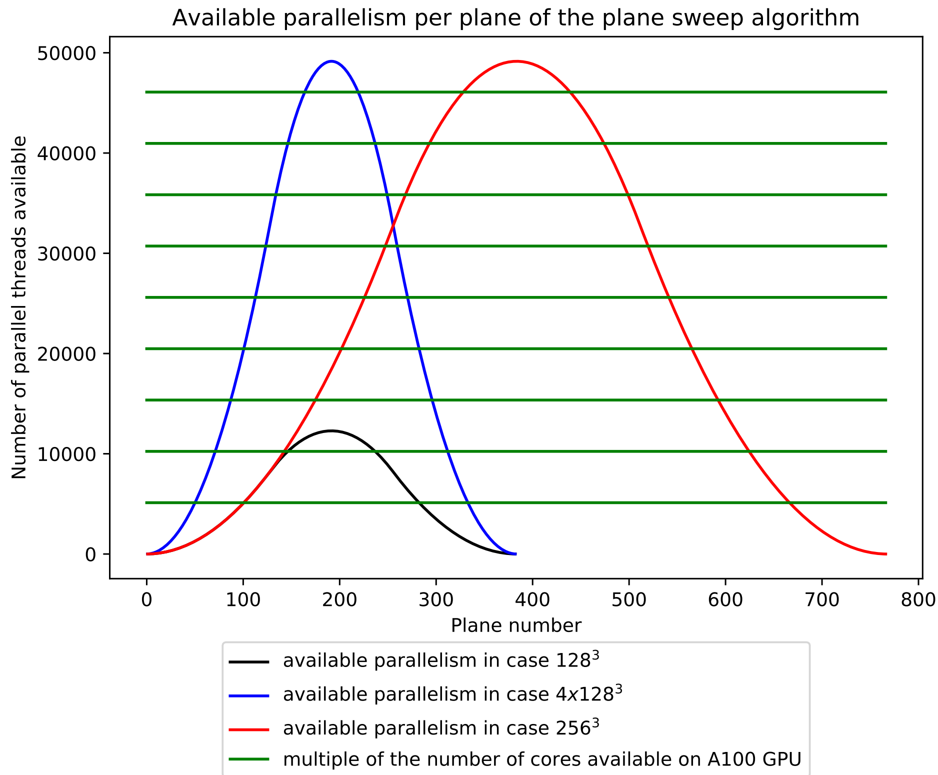


Figure 5.11: Available parallelism for the three smallest plane sweep problems tested for this thesis.

From the graph, it becomes apparent that the smallest problem, 128^3, only fully utilizes all available cores for a fraction of the time, whilst the two bigger problems can fill all cores multiple times over. This means that the larger cases benefit more from the massive parallelism available on a GPU, as theorized in section3.4.
To give a clearer intuition of which part of each problem benefits most from the available parallelism, the theoretical core occupancy is calculated for the smallest three problems. The results are shown in figure 5.12.

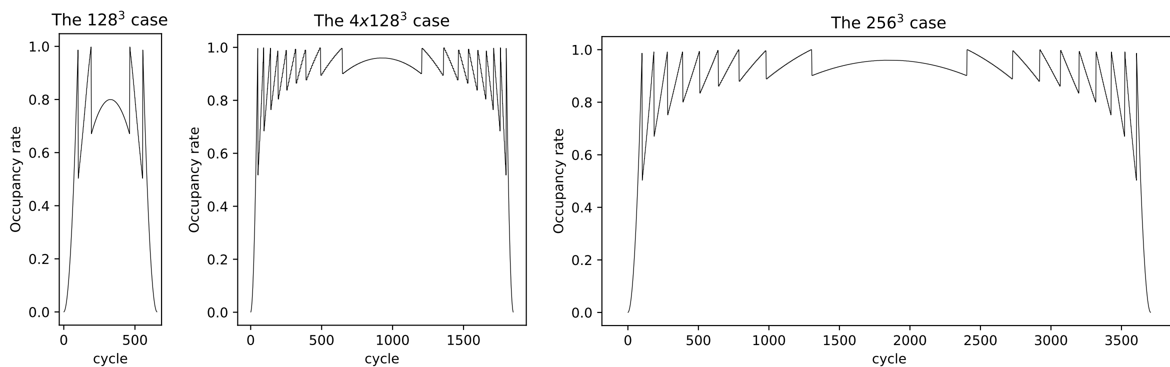


Figure 5.12: Theoretical occupancy rate for each theoretical GPU cycle, plotted for the three smallest plane sweep test problems, all with the same scale.

As becomes clear from the graphs, bigger problems have higher parallelism and as a consequence have a higher occupancy rate on average. This means that larger problems more efficiently make use of the GPU resources, leading to a larger speedup as compared to sequential performance. The 128^3, $4x128^3$ and 256^3 cases of the plane sweep algorithm have an average occupancy rate of 62.6%, 88.6% and 88.4% respectively.

5.4.2. Data transfer time

For the test cases evaluated for this thesis, the data transfer time was significantly larger as compared to the calculation time on the GPU. This means that getting a good estimate of the data transfer time is very important in order to predict the total run time of the GPU accelerated programs. Note that no distinction is made between data transfer time in and out of the GPU.

For the matvec algorithm, using double precision variables (each real number is represented by 8 bytes, each integer by 4), the amount of data transferred D to and from the GPU is estimated to be related to problem size P (in elements) by the relation $D \approx 8$ bytes/number $\cdot$ 8 numbers/element $\cdot$ P. The data quantity and data transfer times for the matvec algorithm are shown in figure 5.13, as well as the modeled data transfer time. For this algorithm, the offset (additional constant due to factors like transfer initialization time) was found to be 0.18 seconds and the effective data transfer rate was found to be 6.2 GB/s. The blue points represent additional measurements to validate the model.

For the plane sweeping algorithm, D can be estimated by the relation:

$$D \approx 8 \text{ bytes/number} \cdot 4 \text{ numbers/element} \cdot P + 4 \text{ bytes/number} \cdot 2 \text{ numbers/element} \cdot N_E,$$

with N_E the number of spatial elements the remark that most plane sweep cases considered are much larger than the matvec cases, as here $P = N_E \cdot N_O$. The data quantity and data transfer times for the plane sweeping algorithm are shown in figures 5.14 and 5.15, as well as the modeled data transfer time. For both grid sizes, the offset was found to be 0.20 seconds and the effective data transfer rate was found to be 7.1 GB/s for the plane sweeping algorithm.

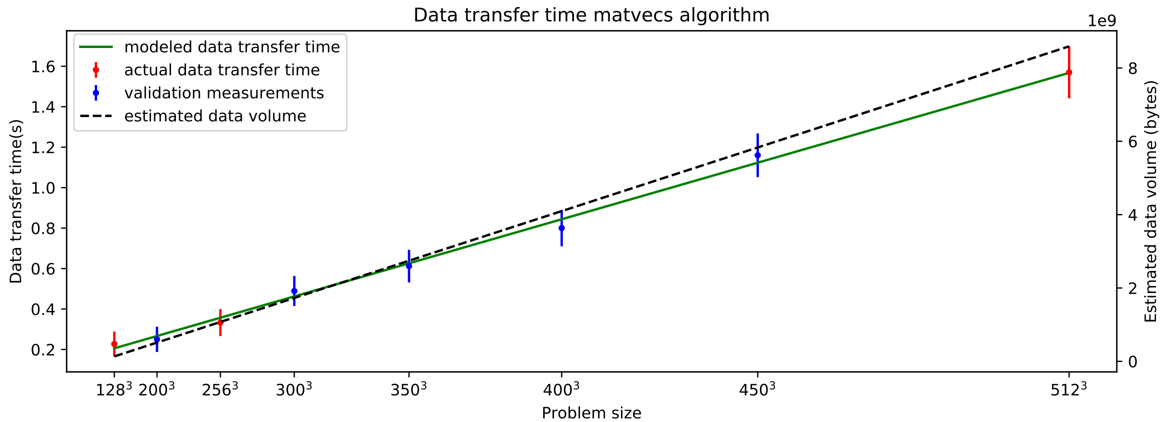


Figure 5.13: Matvec data transfer time.

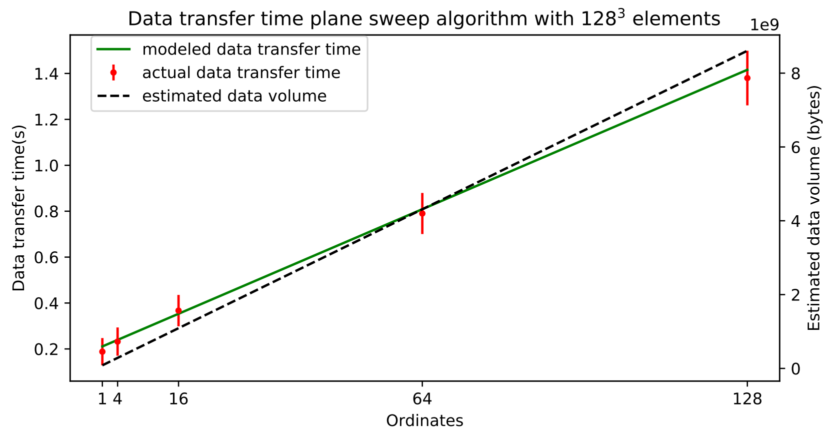


Figure 5.14: Plane sweep data transfer time with 128^3 spatial elements.

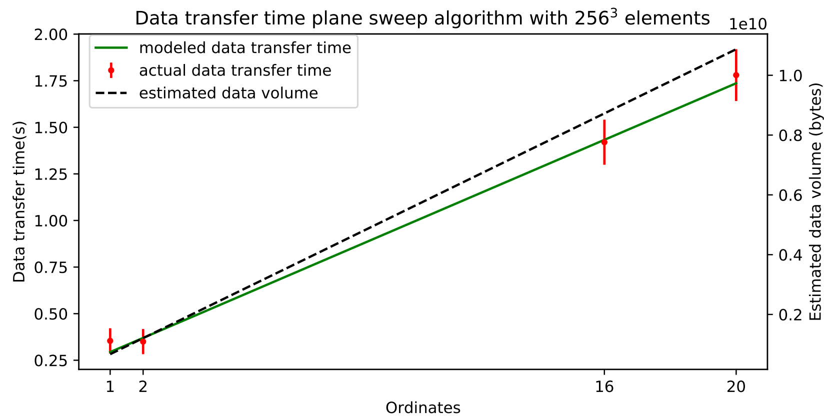


Figure 5.15: Plane sweep data transfer time with 256^3 spatial elements.

5.4.3. Discussion of the model

Because of the scope of this research, the model is only verified for the two algorithms that were considered, the OpenACC implementation in nvfortran, and on the hardware as defined in section 4.2. To expand its validity to more varied problem sets and hardware combinations, more research is needed.

Comparing the parameters found to fit the model to the two different models, it was found that the parameter used to relate $\tilde{T}_s$ to T_s varied a lot between the two applications. More research into this discrepancy is advised to better assess the generality of the model.

6 Conclusion

In this thesis, two algorithms relevant for solving the linear Boltzmann equation for proton irradiation therapy planning applications were accelerated using OpenACC, and a general performance model for the accelerated code was postulated. In this chapter, the general performance model will be repeated, a summary of the main results will be given, as well as an answer to the research question: ”How can GPU offloading decrease computation time for proton therapy dose calculations”. Furthermore, some suggestions for further research are provided.

6.1. The performance model

The general performance model for GPU accelerated code, as deduced in section 3.4, is characterized by the following equation for estimating the parallel execution time T_p:

$$T_p \approx \alpha \frac{T_s}{N_T} \sum_{n=1}^{N} (1 + (N_T(n) - 1)//N_C) + \beta N + \gamma N_T, \tag{6.1}$$

where T_s is the total sequential execution time of the algorithm, N_T is the total number of (parallel) threads the algorithm has available across all regions, with $N_T(n)$ being the number of parallel threads available in the local region, N the number of parallizable regions the code has, and N_C the number of cores available. α, β and γ are fittable constants, as the scaling of this estimator greatly depends on the hardware it is run on and the type of instructions the algorithm consists out of. For the plane sweep algorithm, good fitting constants were found to be 0.250, $3.28 \cdot 10^{-5}$ and $2.11 \cdot 10^{-5}$ for α, β and γ respectively. The $//$ symbol represents the quotient operator and is described in more detail in section 3.4. For the matvec algorithm, a simplified model is used, due to the structure being a single block of many threads:

$$T_p \approx \frac{cT_s}{N_C}, \tag{6.2}$$

with $c = 12$ found to be the best fit to model its performance. More research is advised to asses how the values for these fittable constants could be predicted for an arbitrary algorithm.
Equation 6.1 does not yet take the time it takes to move the required data to and from the GPU into account. This time can be estimated separately dividing the estimated data volume by the effective data transfer rate. The prior can be quite easily calculated, while the latter requires a relatively straightforward test on the hardware the algorithm is intended to be run on.
The model was found to relatively accurately describe the parallel execution time and data transfer time scaling of the accelerated code.

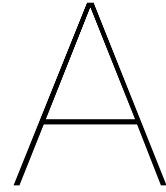
6.2. Acceleration results

The speedup achieved for both algorithms was found to vary with problem parameters, with in general more parallel and larger problems yielding larger speedups. Using OpenACC, both algorithms achieved speedups between 30x and 240x excluding data transfer time, and between 1.3x and 40x including data transfer time, with both values depending on the problem size, with larger problems yielding larger speedups.

If the amount of data transfer is reduced, the speedup achieved including data transfer time increases significantly.

This finally brings us to answering the research question: GPU-offloading can significantly decrease computation time for the main algorithms used in the proton therapy optimization process, with up to 440x improvement on the computation itself.

Further research is advised on how the implementation of those accelerated algorithms in the proton therapy dose calculation algorithm affects its performance, and how data movement in this implementation can be minimized.



Model code

In this chapter of the appendix, the non-accelerated model code is provided. Note that in this chapter, as well as in the next one, some lines were broken off to allow the code to fit on the page.

A.1. Matvec model code

This is the model code for the matvec algorithm without the directives for GPU-offloading. This code was made by taking a finite volume example of the existing algorithm and ”translating” it to the finite element case.

```
program matvec_test
implicit none

double precision, parameter :: Sigma_t=0.1d0
integer, parameter :: ni=500
integer, parameter :: nj=500
integer, parameter :: nk=500

integer, parameter :: no_elem = ni * nj * nk
integer, dimension(no_elem) :: sweep_order
integer, parameter :: no_planes=ni+nj+nk-2
integer, dimension(no_elem) :: plane_number
integer, dimension(no_planes) :: start_index_in_plane
integer, dimension(no_planes) :: end_index_in_plane
double precision, dimension(3) :: Omega
double precision :: phi_in
integer :: i,j,k,elem,plane,index,sweep_elem,nghb,plane_next,elem_next
double precision, dimension(4*no_elem) :: phi,rhs
double precision :: time1,time2
integer :: dim,nghb_start_index,info,start_index
integer :: cr,cm,rate,stime0,stime1,stime2

double precision, dimension(4) :: elem_rhs,temp_vec
double precision, dimension(4) :: Edge_int_xm,Edge_int_ym,Edge_int_zm
double precision, dimension(4,4) :: M,M_xp,M_yp,M_zp
double precision, dimension(4,4) :: M_xm,M_ym,M_zm
double precision, dimension(4,4,3) :: K_mat
double precision, dimension(4,4) :: elem_mat
double precision, dimension(4) :: matvecprod

CALL system_clock(count_rate=cr)
  CALL system_clock(count_max=cm)
```

```
  rate = REAL(cr)

call system_clock(stime0)

! Fill the mass matrix M = \int_V h_i h_j

M = 0.d0
M(1,1) = 1.d0
M(2,2) = 1.d0 / 3.d0
M(3,3) = 1.d0 / 3.d0
M(4,4) = 1.d0 / 3.d0

! Fill the vol matrix K = \int_V grad h_i h_j

K_mat = 0.d0
K_mat(2,1,1) = 2.d0
K_mat(3,1,2) = 2.d0
K_mat(4,1,3) = 2.d0

! Fill M_xp

M_xp = 0.d0
M_xp(1,1) = 1.d0
M_xp(1,2) = 1.d0

M_xp(2,1) = 1.d0
M_xp(2,2) = 1.d0

M_xp(3,3) = 1.d0 / 3.d0

M_xp(4,4) = 1.d0 / 3.d0

! Fill M_yp

M_yp = 0.d0
M_yp(1,1) = 1.d0
M_yp(1,3) = 1.d0

M_yp(2,2) = 1.d0 / 3.d0

M_yp(3,1) = 1.d0
M_yp(3,3) = 1.d0

M_yp(4,4) = 1.d0 / 3.d0

! Fill M_zp

M_zp = 0.d0
M_zp(1,1) = 1.d0
M_zp(1,4) = 1.d0

M_zp(2,2) = 1.d0 / 3.d0

M_zp(3,3) = 1.d0 / 3.d0

M_zp(4,1) = 1.d0
```

```
M_zp(4,4) = 1.d0

! Fill M_xm

M_xm = 0.d0
M_xm(1,1) = 1.d0
M_xm(1,2) = 1.d0

M_xm(2,1) = - 1.d0
M_xm(2,2) = - 1.d0

M_xm(3,3) = 1.d0 / 3.d0

M_xm(4,4) = 1.d0 / 3.d0

! Fill M_ym

M_ym = 0.d0
M_ym(1,1) = 1.d0
M_ym(1,3) = 1.d0

M_ym(2,2) = 1.d0 / 3.d0

M_ym(3,1) = - 1.d0
M_ym(3,3) = - 1.d0

M_ym(4,4) = 1.d0 / 3.d0

! Fill M_zm

M_zm = 0.d0
M_zm(1,1) = 1.d0
M_zm(1,4) = 1.d0

M_zm(2,2) = 1.d0 / 3.d0

M_zm(3,3) = 1.d0 / 3.d0

M_zm(4,1) = - 1.d0
M_zm(4,4) = - 1.d0



! Set the Omega field to a constant

Omega = (/ 1.d0, 1.d0, 1.d0 /)
Omega = Omega / sqrt(dot_product(Omega,Omega))

! Perform sweep
! We assume the volume and areas to be unity

phi = 1.d0
rhs = 0.d0

call system_clock(stime1)
```

```
call cpu_time(time1)

do elem=1,no_elem
  call get_ijk(elem,i,j,k)

  ! Init matrix and rhs

  elem_mat = 0.d0
  elem_rhs = 0.d0

  elem_mat = Sigma_t * M

  ! Volumetric streaming term

    elem_mat(1:4,1:4) = elem_mat(1:4,1:4) - Omega(1) * K_mat(1:4,1:4,1)-
Omega(2) * K_mat(1:4,1:4,2)- Omega(3) * K_mat(1:4,1:4,3)

  ! out x+

  elem_mat = elem_mat + Omega(1) * M_xp

  ! out y+

  elem_mat = elem_mat + Omega(2) * M_yp

  ! out z+

  elem_mat = elem_mat + Omega(3) * M_zp
  ! in x-

  if (i > 1) then
    nghb = get_elem_index(i-1,j,k)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
call matvec(M_xm,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(1) * matvecprod
  else
  endif

  ! in y-

  if (j > 1) then
    nghb = get_elem_index(i,j-1,k)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
call matvec(M_ym,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(2) * matvecprod
  else
  endif

  ! in z-

  if (k > 1) then
    nghb = get_elem_index(i,j,k-1)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
```

```
call matvec(M_zm,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(3) * matvecprod
  else
  endif

  start_index = (elem-1) * 4 + 1
  call matvec(elem_mat,phi(start_index:start_index+4-1),matvecprod)
 rhs(start_index:start_index+4-1) = rhs(start_index:start_index+4-1) + elem_rhs -
matvecprod !matmul(elem_mat,phi(start_index:start_index+4-1))!
enddo

call cpu_time(time2)
call system_clock(stime2)

print *,'Sweep time cpu',time2-time1

print *,'Norm rhs',sqrt(dot_product(rhs,rhs))
print *,'----------------------------------'
print *,'system time:'
print *,'system clock rate',rate
print *,'prep time',real(stime1-stime0)/rate
print *,'sweep time',real(stime2-stime1)/rate


contains
pure subroutine matvec(A,x,y)
!$acc routine seq
implicit none

double precision, intent(in) :: A(4,4),x(4)
double precision, intent(inout) :: y(4)
integer :: i,j

y=0.d0
do i=1,4
do j=1,4
y(i)=y(i)+A(i,j)*x(j)
end do
end do
end subroutine matvec

pure integer function get_elem_index(i,j,k)
implicit none

integer, intent(in) :: i,j,k

get_elem_index = (k-1) * (ni * nj) + (j-1) * (ni) + i

end function get_elem_index



pure subroutine get_ijk(elem,i,j,k)
implicit none

integer, intent(in) :: elem
```

```
integer, intent(out) :: i,j,k

integer :: remainder,remainder_j

k = elem / (ni * nj)
remainder = mod(elem,ni*nj)
if (remainder == 0) then
  k = k
else
  k = k + 1
endif
remainder = elem - (k-1) * (ni*nj)

!!!!!!!

j = remainder / (ni)
remainder_j = mod(remainder,ni)
if (remainder_j == 0) then
  j = j
else
  j = j + 1
endif
remainder = remainder - (j-1) * (ni)

!!!!!!!

i = remainder

end subroutine get_ijk


end program matvec_test
```

A.2. Plane sweep model code

This code is the original model code for the plane sweep algorithm. Note that there are already some OpenMP directives in the code to run the code on four CPU cores. This was used as a starting point in the acceleration process.

```
program sweep_test
use omp_lib
implicit none

! The basis is
! 1, 2x, 2y, 2z. The unit cell is (-1/2,+1/2). So the values are 1 on the bounds

double precision, parameter :: Sigma_t=0.1d0
integer, parameter :: ni=200
integer, parameter :: nj=200
integer, parameter :: nk=200

integer, parameter :: no_elem = ni * nj * nk
integer, dimension(no_elem) :: sweep_order
integer, parameter :: no_planes=ni+nj+nk-2
integer, dimension(no_elem) :: plane_number
integer, dimension(no_planes) :: start_index_in_plane
integer, dimension(no_planes) :: end_index_in_plane
double precision, dimension(3) :: Omega
double precision :: phi_in
integer :: i,j,k,elem,plane,index,sweep_elem,nghb,plane_next,elem_next
double precision, dimension(4 * no_elem) :: phi
double precision :: time1,time2
integer :: dim,nghb_start_index,info,start_index
integer, dimension(4) :: ipiv
double precision, dimension(4,4) :: M,M_xp,M_yp,M_zp
double precision, dimension(4,4) :: M_xm,M_ym,M_zm
double precision, dimension(4,4,3) :: K_mat
double precision, allocatable, dimension(:,:) :: elem_mat
double precision, dimension(4) :: elem_rhs,temp_vec
double precision, dimension(4) :: Edge_int_xm,Edge_int_ym,Edge_int_zm

integer :: qp_x,qp_y,qp_z
double precision, dimension(2) :: points,weights
double precision :: answer,x,y,z

points(1) = +0.5d0 / sqrt(3.d0)
points(2) = -0.5d0 / sqrt(3.d0)
weights(1) = 0.5d0
weights(2) = 0.5d0

! Fill the mass matrix M = \int_V h_i h_j

M = 0.d0
M(1,1) = 1.d0
M(2,2) = 1.d0 / 3.d0
M(3,3) = 1.d0 / 3.d0
M(4,4) = 1.d0 / 3.d0

! Fill the vol matrix K = \int_V grad h_i h_j
```

```
K_mat = 0.d0
K_mat(2,1,1) = 2.d0
K_mat(3,1,2) = 2.d0
K_mat(4,1,3) = 2.d0

! Fill M_xp

M_xp = 0.d0
M_xp(1,1) = 1.d0
M_xp(1,2) = 1.d0

M_xp(2,1) = 1.d0
M_xp(2,2) = 1.d0

M_xp(3,3) = 1.d0 / 3.d0

M_xp(4,4) = 1.d0 / 3.d0

! Fill M_yp

M_yp = 0.d0
M_yp(1,1) = 1.d0
M_yp(1,3) = 1.d0

M_yp(2,2) = 1.d0 / 3.d0

M_yp(3,1) = 1.d0
M_yp(3,3) = 1.d0

M_yp(4,4) = 1.d0 / 3.d0

! Fill M_zp

M_zp = 0.d0
M_zp(1,1) = 1.d0
M_zp(1,4) = 1.d0

M_zp(2,2) = 1.d0 / 3.d0

M_zp(3,3) = 1.d0 / 3.d0

M_zp(4,1) = 1.d0
M_zp(4,4) = 1.d0

! Fill M_xm

M_xm = 0.d0
M_xm(1,1) = 1.d0
M_xm(1,2) = 1.d0

M_xm(2,1) = - 1.d0
M_xm(2,2) = - 1.d0

M_xm(3,3) = 1.d0 / 3.d0

M_xm(4,4) = 1.d0 / 3.d0
```

```
! Fill M_ym

M_ym = 0.d0
M_ym(1,1) = 1.d0
M_ym(1,3) = 1.d0

M_ym(2,2) = 1.d0 / 3.d0

M_ym(3,1) = - 1.d0
M_ym(3,3) = - 1.d0

M_ym(4,4) = 1.d0 / 3.d0

! Fill M_zm

M_zm = 0.d0
M_zm(1,1) = 1.d0
M_zm(1,4) = 1.d0

M_zm(2,2) = 1.d0 / 3.d0

M_zm(3,3) = 1.d0 / 3.d0

M_zm(4,1) = - 1.d0
M_zm(4,4) = - 1.d0

! Fill Edge_int_xm

Edge_int_xm = 0.d0
Edge_int_xm(1) = 1.d0
Edge_int_xm(2) = - 1.d0

! Fill Edge_int_ym

Edge_int_ym = 0.d0
Edge_int_ym(1) = 1.d0
Edge_int_ym(3) = - 1.d0

! Fill Edge_int_zm

Edge_int_zm = 0.d0
Edge_int_zm(1) = 1.d0
Edge_int_zm(4) = - 1.d0

! Find in which sweep plane elem lies

do k=1,nk
  do j=1,nj
    do i=1,ni
      elem = get_elem_index(i,j,k)
      plane_number(elem) = i+j+k-2
    enddo
  enddo
enddo
```

```
! Now do a very poor implementation of getting then sweep order

index=1
do plane=1,no_planes
  do elem=1,no_elem
    if (plane_number(elem) == plane) then
      sweep_order(index) = elem
      index = index + 1
    endif
  enddo
enddo

! Get the start and end indices for each plane

start_index_in_plane(1) = 1
do sweep_elem=1,no_elem-1
  elem = sweep_order(sweep_elem)
  elem_next = sweep_order(sweep_elem+1)

  plane = plane_number(elem)
  plane_next = plane_number(elem_next)

  if (plane_next /= plane) then
    end_index_in_plane(plane) = sweep_elem
    start_index_in_plane(plane+1) = sweep_elem+1
  endif
enddo
end_index_in_plane(no_planes) = no_elem

! Set the Omega field to a constant

Omega = (/ 1.d0, 1.d0, 1.d0 /)
Omega = Omega / sqrt(dot_product(Omega,Omega))

! Perform sweep
! We assume the volume and areas to be unity

phi = 0.d0

call cpu_time(time1)
do plane=1,no_planes
!$OMP target teams distribute parallel &
!$OMP default(none) &
!$OMP private(start_index,nghb_start_index,elem,i,j,k,info,ipiv,elem_mat,
elem_rhs,nghb,phi_in,temp_vec,sweep_elem) &
!$OMP shared(phi,start_index_in_plane,end_index_in_plane,Omega,K_mat,M,plane,
sweep_order,M_xp,M_yp,M_zp,M_xm,M_ym,M_zm,Edge_int_xm,Edge_int_ym,Edge_int_zm) &
!OMP num_threads(4)
!$OMP do schedule(static)
do sweep_elem=start_index_in_plane(plane),end_index_in_plane(plane)
  elem = sweep_order(sweep_elem)
  call get_ijk(elem,i,j,k)

  ! Init matrix and rhs

  allocate(elem_mat(4,4))
```

```
elem_mat = 0.d0
elem_rhs = 0.d0

! Removal

elem_mat = Sigma_t * M

! Volumetric streaming term

do dim=1,3
  elem_mat(1:4,1:4) = elem_mat(1:4,1:4) - Omega(dim) * K_mat(1:4,1:4,dim)
enddo

! out x+

elem_mat = elem_mat + Omega(1) * M_xp

! out y+

elem_mat = elem_mat + Omega(2) * M_yp

! out z+

elem_mat = elem_mat + Omega(3) * M_zp

! in x-

if (i > 1) then
  nghb = get_elem_index(i-1,j,k)
  nghb_start_index = (nghb-1) * 4 + 1
  temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
  elem_rhs = elem_rhs + Omega(1) * matmul(M_xm,temp_vec)
else
  phi_in = 1.d0
  elem_rhs = elem_rhs + phi_in * Omega(1) * Edge_int_xm
endif

! in y-

if (j > 1) then
  nghb = get_elem_index(i,j-1,k)
  nghb_start_index = (nghb-1) * 4 + 1
  temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
  elem_rhs = elem_rhs + Omega(2) * matmul(M_ym,temp_vec)
else
  phi_in = 1.d0
  elem_rhs = elem_rhs + phi_in * Omega(2) * Edge_int_ym
endif

! in z-

if (k > 1) then
  nghb = get_elem_index(i,j,k-1)
  nghb_start_index = (nghb-1) * 4 + 1
  temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
  elem_rhs = elem_rhs + Omega(3) * matmul(M_zm,temp_vec)
```

```
  else
    phi_in = 1.d0
    elem_rhs = elem_rhs + phi_in * Omega(3) * Edge_int_zm
  endif

  ! Local solve

  call dgesv(4,1,elem_mat,4,ipiv,elem_rhs,4,info)
  if (info /= 0) STOP 'problem in dgesv'
  start_index = (elem-1) * 4 + 1
  phi(start_index:start_index+4-1) = elem_rhs

  deallocate(elem_mat)
enddo
!$OMP end do
!$OMP end target teams distribute parallel
enddo
call cpu_time(time2)
print *,'Sweep time',time2-time1

print *,'Norm phi',sqrt(dot_product(phi,phi))

contains

integer function get_elem_index(i,j,k)
implicit none

integer :: i,j,k

get_elem_index = (k-1) * (ni * nj) + (j-1) * (ni) + i

end function get_elem_index



subroutine get_ijk(elem,i,j,k)
implicit none

integer :: elem,i,j,k

integer :: remainder,remainder_j

k = elem / (ni * nj)
remainder = mod(elem,ni*nj)
if (remainder == 0) then
  k = k
else
  k = k + 1
endif
remainder = elem - (k-1) * (ni*nj)

!!!!!!!

j = remainder / (ni)
remainder_j = mod(remainder,ni)
if (remainder_j == 0) then
```

```
  j = j
else
  j = j + 1
endif
remainder = remainder - (j-1) * (ni)

!!!!!!!

i = remainder

end subroutine get_ijk


double precision function fi(i,x,y,z)
implicit none

integer :: i
double precision :: x,y,z

if (i==1) then
  fi = 1.d0
endif
if (i==2) then
  fi = 2.d0 * x
endif
if (i==3) then
  fi = 2.d0 * y
endif
if (i==4) then
  fi = 2.d0 * z
endif

end function fi


double precision function grad_fi(i,dim,x,y,z)
implicit none

integer :: i,dim
double precision :: x,y,z

grad_fi = 0.d0
if (i==2) then
  if (dim==1) grad_fi = 2.d0
endif
if (i==3) then
  if (dim==2) grad_fi = 2.d0
endif
if (i==4) then
  if (dim==3) grad_fi = 2.d0
endif

end function grad_fi

end program sweep_test
```

B
Accelerated code

This chapter of the appendix contains the accelerated model code as created during this project. Note that, like in the previous chapter, some lines of code were broken off in order to fit on the page.

B.1. Accelerated matvec model code

This code is the accelerated version of the model code provided in appendix A.1. Data and parallel loop directives have been added to make it run in parallel.

```
program matvec_test
implicit none

double precision, parameter :: Sigma_t=0.1d0
integer, parameter :: ni=500
integer, parameter :: nj=500
integer, parameter :: nk=500

integer, parameter :: no_elem = ni * nj * nk
integer, dimension(no_elem) :: sweep_order
integer, parameter :: no_planes=ni+nj+nk-2
integer, dimension(no_elem) :: plane_number
integer, dimension(no_planes) :: start_index_in_plane
integer, dimension(no_planes) :: end_index_in_plane
double precision, dimension(3) :: Omega
double precision :: phi_in
integer :: i,j,k,elem,plane,index,sweep_elem,nghb,plane_next,elem_next
double precision, dimension(4*no_elem) :: phi,rhs
double precision :: time1,time2
integer :: dim,nghb_start_index,info,start_index
integer :: cr,cm,rate,stime0,stimepredat,stime1,stime2,stime3

double precision, dimension(4) :: elem_rhs,temp_vec
double precision, dimension(4) :: Edge_int_xm,Edge_int_ym,Edge_int_zm
double precision, dimension(4,4) :: M,M_xp,M_yp,M_zp
double precision, dimension(4,4) :: M_xm,M_ym,M_zm
double precision, dimension(4,4,3) :: K_mat
double precision, dimension(4,4) :: elem_mat
double precision, dimension(4) :: matvecprod

CALL system_clock(count_rate=cr)
  CALL system_clock(count_max=cm)
```

```
  rate = REAL(cr)

call system_clock(stime0)

! Fill the mass matrix M = \int_V h_i h_j

M = 0.d0
M(1,1) = 1.d0
M(2,2) = 1.d0 / 3.d0
M(3,3) = 1.d0 / 3.d0
M(4,4) = 1.d0 / 3.d0

! Fill the vol matrix K = \int_V grad h_i h_j

K_mat = 0.d0
K_mat(2,1,1) = 2.d0
K_mat(3,1,2) = 2.d0
K_mat(4,1,3) = 2.d0

! Fill M_xp

M_xp = 0.d0
M_xp(1,1) = 1.d0
M_xp(1,2) = 1.d0

M_xp(2,1) = 1.d0
M_xp(2,2) = 1.d0

M_xp(3,3) = 1.d0 / 3.d0

M_xp(4,4) = 1.d0 / 3.d0

! Fill M_yp

M_yp = 0.d0
M_yp(1,1) = 1.d0
M_yp(1,3) = 1.d0

M_yp(2,2) = 1.d0 / 3.d0

M_yp(3,1) = 1.d0
M_yp(3,3) = 1.d0

M_yp(4,4) = 1.d0 / 3.d0

! Fill M_zp

M_zp = 0.d0
M_zp(1,1) = 1.d0
M_zp(1,4) = 1.d0

M_zp(2,2) = 1.d0 / 3.d0

M_zp(3,3) = 1.d0 / 3.d0

M_zp(4,1) = 1.d0
```

```
M_zp(4,4) = 1.d0

! Fill M_xm

M_xm = 0.d0
M_xm(1,1) = 1.d0
M_xm(1,2) = 1.d0

M_xm(2,1) = - 1.d0
M_xm(2,2) = - 1.d0

M_xm(3,3) = 1.d0 / 3.d0

M_xm(4,4) = 1.d0 / 3.d0

! Fill M_ym

M_ym = 0.d0
M_ym(1,1) = 1.d0
M_ym(1,3) = 1.d0

M_ym(2,2) = 1.d0 / 3.d0

M_ym(3,1) = - 1.d0
M_ym(3,3) = - 1.d0

M_ym(4,4) = 1.d0 / 3.d0

! Fill M_zm

M_zm = 0.d0
M_zm(1,1) = 1.d0
M_zm(1,4) = 1.d0

M_zm(2,2) = 1.d0 / 3.d0

M_zm(3,3) = 1.d0 / 3.d0

M_zm(4,1) = - 1.d0
M_zm(4,4) = - 1.d0




! Set the Omega field to a constant

Omega = (/ 1.d0, 1.d0, 1.d0 /)
Omega = Omega / sqrt(dot_product(Omega,Omega))

! Perform sweep
! We assume the volume and areas to be unity

phi = 1.d0
rhs = 0.d0
call system_clock(stimepredat)
!$acc data
```

```
!$acc enter data copyin(phi), copyin(rhs),
copyin(omega,M,m_yp,m,k_mat,m_zp,m_xm,m_ym,m_zm,m_xp)
call system_clock(stime1)
call cpu_time(time1)

!$acc parallel loop gang private(elem_mat,elem_rhs,nghb,nghb_start_index,
temp_vec,matvecprod) default(present)
do elem=1,no_elem
  call get_ijk(elem,i,j,k)

  ! Init matrix and rhs

  elem_mat = 0.d0
  elem_rhs = 0.d0

  elem_mat = Sigma_t * M

  ! Volumetric streaming term

    elem_mat(1:4,1:4) = elem_mat(1:4,1:4) - Omega(1) * K_mat(1:4,1:4,1)-
Omega(2) * K_mat(1:4,1:4,2)- Omega(3) * K_mat(1:4,1:4,3)

  ! out x+

  elem_mat = elem_mat + Omega(1) * M_xp

  ! out y+

  elem_mat = elem_mat + Omega(2) * M_yp

  ! out z+

  elem_mat = elem_mat + Omega(3) * M_zp
  ! in x-

  if (i > 1) then
    nghb = get_elem_index(i-1,j,k)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
call matvec(M_xm,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(1) * matvecprod
  else
  endif

  ! in y-

  if (j > 1) then
    nghb = get_elem_index(i,j-1,k)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
call matvec(M_ym,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(2) * matvecprod
  else
  endif

  ! in z-
```

```
  if (k > 1) then
    nghb = get_elem_index(i,j,k-1)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1)
call matvec(M_zm,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(3) * matvecprod
  else
  endif

  start_index = (elem-1) * 4 + 1
  call matvec(elem_mat,phi(start_index:start_index+4-1),matvecprod)
 rhs(start_index:start_index+4-1) = rhs(start_index:start_index+4-1) + elem_rhs -
matvecprod !matmul(elem_mat,phi(start_index:start_index+4-1))!
enddo
!$acc end parallel loop

call cpu_time(time2)
call system_clock(stime2)
!$acc exit data copyout(rhs)
!$acc end data
call system_clock(stime3)
print *,'Sweep time cpu',time2-time1

print *,'Norm rhs',sqrt(dot_product(rhs,rhs))
print *,'----------------------------------'
print *,'system time:'
print *,'system clock rate',rate
print *,'prep time',real(stimepredat-stime0)/rate
print *,'copyin time',real(stime1-stimepredat)/rate
print *,'sweep time',real(stime2-stime1)/rate
print *,'copyout time',real(stime3-stime2)/rate


contains
pure subroutine matvec(A,x,y)
!$acc routine seq
implicit none

double precision, intent(in) :: A(4,4),x(4)
double precision, intent(inout) :: y(4)
integer :: i,j

y=0.d0
do i=1,4
do j=1,4
y(i)=y(i)+A(i,j)*x(j)
end do
end do
end subroutine matvec

pure integer function get_elem_index(i,j,k)
implicit none

integer, intent(in) :: i,j,k
```

```
get_elem_index = (k-1) * (ni * nj) + (j-1) * (ni) + i

end function get_elem_index



pure subroutine get_ijk(elem,i,j,k)
implicit none

integer, intent(in) :: elem
integer, intent(out) :: i,j,k

integer :: remainder,remainder_j

k = elem / (ni * nj)
remainder = mod(elem,ni*nj)
if (remainder == 0) then
  k = k
else
  k = k + 1
endif
remainder = elem - (k-1) * (ni*nj)

!!!!!!!

j = remainder / (ni)
remainder_j = mod(remainder,ni)
if (remainder_j == 0) then
  j = j
else
  j = j + 1
endif
remainder = remainder - (j-1) * (ni)

!!!!!!!

i = remainder

end subroutine get_ijk


end program matvec_test
```

B.2. Accelerated plane sweep model code

This code is the accelerated version of the model code provided in appendix A.2. A few things have been added: first of all the parallel loop directives as mentioned before. Additionally, data directives have been added, as well as some additional commands and infrastructure adaptations to allow the parallel execution of multiple discrete directions. Another change as compared to the original model code, was the replacement of the dgesv function from the LaPack library with a custom made sovler, and the replacement of the matmul intrinsic with a dedicated matvec function for multiplying vectors with matrices.

```
program sweep_test
implicit none

! The basis is
! 1, 2x, 2y, 2z. The unit cell is (-1/2,+1/2). So the values are 1 on the bounds

double precision, parameter :: Sigma_t=0.1d0
integer, parameter :: ni=128
integer, parameter :: nj=128
integer, parameter :: nk=128

integer, parameter :: N= 16!N is the number of directional sweep vectors
integer :: O(N) !O denotes the ordinate belonging to a given vector omega
integer :: dir  !extra loopiterator introduced for looping over different omega vectors

integer, parameter :: no_elem = ni * nj * nk
integer, dimension(no_elem) :: sweep_order
integer, parameter :: no_planes=ni+nj+nk-2
integer, dimension(no_elem) :: plane_number
integer, dimension(no_planes) :: start_index_in_plane
integer, dimension(no_planes) :: end_index_in_plane
double precision, dimension(3,N) :: Omega
!Omega is now a matrix with N Omega vectors as its columns
double precision :: phi_in
integer :: i,j,k,elem,plane,index,sweep_elem,nghb,plane_next,elem_next
double precision, dimension(4 * no_elem,N) :: phi
!phi is now a matrix with N phi vectors as its columns
double precision :: time1,time2
integer :: dim,nghb_start_index,start_index
double precision, dimension(4,4) :: M,M_xp,M_yp,M_zp
double precision, dimension(4,4) :: M_xm,M_ym,M_zm
double precision, dimension(4,4,3) :: K_mat
double precision, dimension(4,4) :: elem_mat
double precision, dimension(4) :: elem_rhs,temp_vec,matvecprod
double precision, dimension(4) :: Edge_int_xm,Edge_int_ym,Edge_int_zm

integer :: qp_x,qp_y,qp_z
double precision, dimension(2) :: points,weights
double precision :: answer,x,y,z

integer :: cr,cm,rate,stime0,stimepredat,stime1,stime2,stime3,stime15

CALL system_clock(count_rate=cr)
  CALL system_clock(count_max=cm)
  rate = REAL(cr)
```

```
call system_clock(stime0)

points(1) = +0.5d0 / sqrt(3.d0)
points(2) = -0.5d0 / sqrt(3.d0)
weights(1) = 0.5d0
weights(2) = 0.5d0

! Fill the mass matrix M = \int_V h_i h_j

M = 0.d0
M(1,1) = 1.d0
M(2,2) = 1.d0 / 3.d0
M(3,3) = 1.d0 / 3.d0
M(4,4) = 1.d0 / 3.d0

! Fill the vol matrix K = \int_V grad h_i h_j

K_mat = 0.d0
K_mat(2,1,1) = 2.d0
K_mat(3,1,2) = 2.d0
K_mat(4,1,3) = 2.d0

! Fill M_xp

M_xp = 0.d0
M_xp(1,1) = 1.d0
M_xp(1,2) = 1.d0

M_xp(2,1) = 1.d0
M_xp(2,2) = 1.d0

M_xp(3,3) = 1.d0 / 3.d0

M_xp(4,4) = 1.d0 / 3.d0

! Fill M_yp

M_yp = 0.d0
M_yp(1,1) = 1.d0
M_yp(1,3) = 1.d0

M_yp(2,2) = 1.d0 / 3.d0

M_yp(3,1) = 1.d0
M_yp(3,3) = 1.d0

M_yp(4,4) = 1.d0 / 3.d0

! Fill M_zp

M_zp = 0.d0
M_zp(1,1) = 1.d0
M_zp(1,4) = 1.d0

M_zp(2,2) = 1.d0 / 3.d0
```

```
M_zp(3,3) = 1.d0 / 3.d0

M_zp(4,1) = 1.d0
M_zp(4,4) = 1.d0

! Fill M_xm

M_xm = 0.d0
M_xm(1,1) = 1.d0
M_xm(1,2) = 1.d0

M_xm(2,1) = - 1.d0
M_xm(2,2) = - 1.d0

M_xm(3,3) = 1.d0 / 3.d0

M_xm(4,4) = 1.d0 / 3.d0

! Fill M_ym

M_ym = 0.d0
M_ym(1,1) = 1.d0
M_ym(1,3) = 1.d0

M_ym(2,2) = 1.d0 / 3.d0

M_ym(3,1) = - 1.d0
M_ym(3,3) = - 1.d0

M_ym(4,4) = 1.d0 / 3.d0

! Fill M_zm

M_zm = 0.d0
M_zm(1,1) = 1.d0
M_zm(1,4) = 1.d0

M_zm(2,2) = 1.d0 / 3.d0

M_zm(3,3) = 1.d0 / 3.d0

M_zm(4,1) = - 1.d0
M_zm(4,4) = - 1.d0

! Fill Edge_int_xm

Edge_int_xm = 0.d0
Edge_int_xm(1) = 1.d0
Edge_int_xm(2) = - 1.d0

! Fill Edge_int_ym

Edge_int_ym = 0.d0
Edge_int_ym(1) = 1.d0
Edge_int_ym(3) = - 1.d0
```

```
! Fill Edge_int_zm

Edge_int_zm = 0.d0
Edge_int_zm(1) = 1.d0
Edge_int_zm(4) = - 1.d0

! Find in which sweep plane elem lies

do k=1,nk
  do j=1,nj
    do i=1,ni
      elem = get_elem_index(i,j,k)
      plane_number(elem) = i+j+k-2
    enddo
  enddo
enddo

!Now get the sweep order

index=1
do plane=1,no_planes
  do elem=1,no_elem
    if (plane_number(elem) == plane) then
      sweep_order(index) = elem
      index = index + 1
    endif
  enddo
enddo

! Get the start and end indices for each plane

start_index_in_plane(1) = 1
do sweep_elem=1,no_elem-1
  elem = sweep_order(sweep_elem)
  elem_next = sweep_order(sweep_elem+1)

  plane = plane_number(elem)
  plane_next = plane_number(elem_next)

  if (plane_next /= plane) then
    end_index_in_plane(plane) = sweep_elem
    start_index_in_plane(plane+1) = sweep_elem+1
  endif
enddo
end_index_in_plane(no_planes) = no_elem
! Set the Omega field to a constant
Omega = 1.d0 !to ensure the code doesn't break when larger N is tested
Omega(:,1) = (/ 1.d0, 1.d0, 1.d0 /)
!Omega definition was adjusted to allow definition of N different vectors
distributed across the 8 octants
Omega(:,2) = (/ 1.d0, -1.d0, 5.d0 /)

do dir=1,N !normalisation step adjusted to make sure each Omega vector has length one
Omega(:,dir) = Omega(:,dir) / sqrt(dot_product(Omega(:,dir),Omega(:,dir)))
enddo
```

```
! Perform sweep
! We assume the volume and areas to be unity

phi = 0.d0


call system_clock(stimepredat)
!$acc data copyin(sweep_order,start_index,start_index_in_plane,end_index_in_plane,M,
Omega,K_mat,M_xp,M_yp,M_zp,M_xm,M_ym,M_zm,Edge_int_xm,Edge_int_ym,Edge_int_zm)
copyout(phi,O)
create(matvecprod,nghb,nghb_start_index,elem,i,j,k,elem_mat,
elem_rhs,temp_vec,phi_in,sweep_elem,dir)
call system_clock(stime1)
!while we’re already at the gpu, let’s determine the octant of each omega and make sure
to adapt omega to the original range to not break the code already present
!$acc parallel loop gang num_gangs(N) private(dir) present(Omega) default(none)
do dir=1,N
O(dir) = getO(Omega(:,dir))
!assigns octant number to omega associated with current direction
!the following three lines mirrors negative coordinates for each direction,
because the original code is not equiped to handle negative definite omega vectors
Omega(1,dir) = abs(Omega(1,dir))
Omega(2,dir) = abs(Omega(2,dir))
Omega(3,dir) = abs(Omega(3,dir))
enddo
call system_clock(stime15)
call cpu_time(time1)
do plane=1,no_planes
!$acc parallel loop gang vector collapse(2) default(none)
private(start_index,nghb_start_index,elem,i,j,k,elem_mat,
elem_rhs,nghb,phi_in,temp_vec,sweep_elem,dir)
present(M_zm,M_ym,M_xm,M_zp,M_yp,M_xp,K_mat,Omega,M,sweep_order,phi,Edge_int_xm,
Edge_int_ym,Edge_int_zm,end_index_in_plane,start_index_in_plane)
do dir=1,N !Note that this loop is nested in the planes loop.
This is because only gang level parallelism allows for private variables,
which is required to generate correct results.
Gang parallelism does not work for nested loop directives,
which is why instead both parallel loops are grouped together and collapsed.
do sweep_elem=start_index_in_plane(plane),end_index_in_plane(plane)
  elem = sweep_order(sweep_elem)
  call get_ijk(elem,i,j,k)
  ! Init matrix and rhs
  elem_mat = 0.d0
  elem_rhs = 0.d0

  ! Removal

  elem_mat = Sigma_t * M

  ! Volumetric streaming term
!in the following lines, Omega and phi slicing were adapted
to account for the new structure of these variables
   elem_mat(1:4,1:4) = elem_mat(1:4,1:4) - Omega(1,dir) * K_mat(1:4,1:4,1)-
Omega(2,dir) * K_mat(1:4,1:4,2)- Omega(3,dir) * K_mat(1:4,1:4,3)

  ! out x+
```

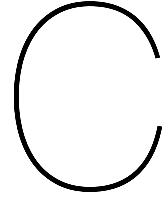
```
  elem_mat = elem_mat + Omega(1,dir) * M_xp

  ! out y+

  elem_mat = elem_mat + Omega(2,dir) * M_yp

  ! out z+

  elem_mat = elem_mat + Omega(3,dir) * M_zp
  ! in x-

  if (i > 1) then
    nghb = get_elem_index(i-1,j,k)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1,dir)
call matvec(M_xm,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(1,dir) * matvecprod
  else
    phi_in = 1.d0
    elem_rhs = elem_rhs + phi_in * Omega(1,dir) * Edge_int_xm
  endif

  ! in y-

  if (j > 1) then
    nghb = get_elem_index(i,j-1,k)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1,dir)
call matvec(M_ym,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(2,dir) * matvecprod
  else
    phi_in = 1.d0
    elem_rhs = elem_rhs + phi_in * Omega(2,dir) * Edge_int_ym
  endif

  ! in z-

  if (k > 1) then
    nghb = get_elem_index(i,j,k-1)
    nghb_start_index = (nghb-1) * 4 + 1
    temp_vec = phi(nghb_start_index:nghb_start_index+4-1,dir)
call matvec(M_zm,temp_vec,matvecprod)
    elem_rhs = elem_rhs + Omega(3,dir) * matvecprod
  else
    phi_in = 1.d0
    elem_rhs = elem_rhs + phi_in * Omega(3,dir) * Edge_int_zm
  endif

  call dgesv_gpu(4,elem_mat,elem_rhs)
  start_index = (elem-1) * 4 + 1
  phi(start_index:start_index+4-1,dir) = elem_rhs

enddo
enddo
enddo
```

```
call system_clock(stime2)
!$acc end data
call cpu_time(time2)
call system_clock(stime3)
print *,'Sweep time',time2-time1

print *,'Sum of phinorms',sumnormphi(phi,N)
!here the phinorm calculation was adapted to give a single variable combining
all different directions that were calculated
print *,'octant of each vector omega',O(:)


print *,'----------------------------------'
!adapted system time to account for contribution ordinate calculation time
print *,'system time:'
print *,'system clock rate',rate
print *,'prep time',real(stimepredat-stime0)/rate
print *,'copyin time',real(stime1-stimepredat)/rate
print *,'ordinate calculation time',real(stime15-stime1)/rate
print *,'sweep time',real(stime2-stime15)/rate
print *,'copyout time',real(stime3-stime2)/rate


contains
pure subroutine matvec(A,x,y)
!$acc routine seq
implicit none

double precision, intent(in) :: A(4,4),x(4)
double precision, intent(inout) :: y(4)
integer :: i,j

y=0.d0
do i=1,4
do j=1,4
y(i)=y(i)+A(i,j)*x(j)
end do
end do
end subroutine matvec


double precision function sumnormphi(phi,N) !added to check correctness between
cpu and gpu, replaces norm of phi in original output
!$acc routine seq
implicit none
double precision :: phi(:,:)
integer :: direc,N
sumnormphi = 0.d0
do direc=1,N
sumnormphi = sumnormphi+sqrt(dot_product(phi(:,direc),phi(:,direc)))
enddo
end function sumnormphi

pure integer function getO(Omega) !added to calculate the octant of each omega vector,
which is relevant to keep track of the original sweep direction in a heterogenous
```

```
medium and the orientation of the phi contribution corresponding to this omega
!$acc routine seq
implicit none
double precision,dimension(3),intent(in) :: Omega
getO = int(2*(1-sign(1.d0,Omega(3)))+(1-sign(1.d0,Omega(2)))
+0.5*(1-sign(1.d0,Omega(3))))
end function getO

integer function get_elem_index(i,j,k)
!$acc routine seq
implicit none

integer :: i,j,k

get_elem_index = (k-1) * (ni * nj) + (j-1) * (ni) + i

end function get_elem_index



subroutine get_ijk(elem,i,j,k)
!$acc routine seq
implicit none

integer :: elem,i,j,k

integer :: remainder,remainder_j

k = elem / (ni * nj)
remainder = mod(elem,ni*nj)
if (remainder == 0) then
  k = k
else
  k = k + 1
endif
remainder = elem - (k-1) * (ni*nj)

!!!!!!!

j = remainder / (ni)
remainder_j = mod(remainder,ni)
if (remainder_j == 0) then
  j = j
else
  j = j + 1
endif
remainder = remainder - (j-1) * (ni)

!!!!!!!

i = remainder

end subroutine get_ijk


double precision function fi(i,x,y,z)
```

```
!$acc routine seq
implicit none

integer :: i
double precision :: x,y,z

if (i==1) then
  fi = 1.d0
endif
if (i==2) then
  fi = 2.d0 * x
endif
if (i==3) then
  fi = 2.d0 * y
endif
if (i==4) then
  fi = 2.d0 * z
endif

end function fi


double precision function grad_fi(i,dim,x,y,z)
!$acc routine seq
implicit none

integer :: i,dim
double precision :: x,y,z

grad_fi = 0.d0
if (i==2) then
  if (dim==1) grad_fi = 2.d0
endif
if (i==3) then
  if (dim==2) grad_fi = 2.d0
endif
if (i==4) then
  if (dim==3) grad_fi = 2.d0
endif

end function grad_fi

 pure subroutine dgesv_gpu(n,A,b) !
!$acc routine seq
implicit none

integer, intent(in) :: n
double precision, intent(inout) :: A(4,4),b(4)

integer :: i,row,col,ind
double precision  :: p,t
double precision  :: maximum !added to replace maxloc

do row=1,n-1
  ! Partial pivotting: (the next 6 lines have been added because maxloc is not allowed)
```

```
  maximum = abs(A(row,row))
  ind=row
  do i=row+1,n
    if (abs(A(i,row)) > maximum) then
      maximum = abs(A(i,row))
      ind = i
    endif
  enddo

  ! Swap rows of A and b

  if (ind /= row) then
    do col=row,n
      t = A(ind,col)
      A(ind,col) = A(row,col)
      A(row,col) = t
    enddo
    t = b(ind)
    b(ind) = b(row)
    b(row) = t
  endif

  ! Eliminate by row subtraction

  do i=row+1,n
    p = A(i,row) / A(row,row)
    A(i,row+1:n) = A(i,row+1:n) - p * A(row,row+1:n)
    b(i)         = b(i)         - p * b(row)
  enddo
enddo

! Back substitution

do row=n,1,-1
  p = b(row)
  do col=row+1,n
    p = p - A(row,col) * b(col)
  end do
  b(row) = p / A(row,row)
enddo

end subroutine dgesv_gpu


end program sweep_test
```



Performance measurement tables

C.1. Matvec

hardware	com+op	size	copy in	sweep time	copy out	data	combined
gpu	nv pinned	128	2.19E-01	5.60E-04	1.01E-02	2.29E-01	2.30E-01
cpu	gf	128	0.00E+00	2.86E-01	0.00E+00	0.00E+00	2.86E-01
cpu	ifx	128	0.00E+00	1.46E-01	0.00E+00	0.00E+00	1.46E-01
cpu	nv	128	0.00E+00	2.00E-01	0.00E+00	0.00E+00	2.00E-01
gpu	nv	128	2.17E-01	5.56E-04	1.00E-02	2.27E-01	2.28E-01
gpu	nv pinned	256	2.55E-01	4.11E-03	7.44E-02	3.30E-01	3.34E-01
gpu	nv	256	2.59E-01	4.12E-03	7.44E-02	3.33E-01	3.37E-01
cpu	nv	256	0.00E+00	1.53E+00	0.00E+00	0.00E+00	1.53E+00
cpu	gf	256	0.00E+00	2.24E+00	0.00E+00	0.00E+00	2.24E+00
cpu	ifx	256	0.00E+00	1.18E+00	0.00E+00	0.00E+00	1.18E+00
cpu	nv multic	256	0.00E+00	7.88E-02	0.00E+00	0.00E+00	7.88E-02
gpu	nv pinned	512	1.20E+00	2.63E-02	5.97E-01	1.80E+00	1.82E+00
gpu	nv	512	8.86E-01	2.63E-02	6.89E-01	1.57E+00	1.60E+00
cpu	nv	512	0.00E+00	1.15E+01	0.00E+00	0.00E+00	1.15E+01
gpu	nv	200	2.14E-01	1.72E-03	3.65E-02	2.50E-01	2.52E-01
gpu	nv	300	3.65E-01	6.57E-03	1.24E-01	4.89E-01	4.96E-01
gpu	nv	350	4.13E-01	9.07E-03	1.99E-01	6.12E-01	6.21E-01
gpu	nv	450	7.15E-01	1.91E-02	4.44E-01	1.16E+00	1.18E+00
gpu	nv	400	5.08E-01	1.32E-02	2.92E-01	8.00E-01	8.13E-01

Table C.1: Performance data of the matvec algorithm. The first column from the right shows the time including data transfer and the data column shows the total data transfer time. The sweep time column shows the completion time without data transfers. The second column shows the compiler that is used, including special options.

hardware	com+op	size	norm_rhs
gpu	nv pinned	128	2.9929513675E+03
cpu	gf	128	2.9929513675E+03
cpu	ifx	128	2.9929513675E+03
cpu	nv	128	2.9929513675E+03
gpu	nv	128	2.9929513675E+03
gpu	nv pinned	256	8.4518137372E+03
gpu	nv	256	8.4518137372E+03
cpu	nv	256	8.4518137372E+03
cpu	gf	256	8.4518137372E+03
cpu	ifx	256	8.4518137385E+03
cpu	nv multic	256	8.4518137372E+03
gpu	nv pinned	512	2.3886358925E+04
gpu	nv	512	2.3886358925E+04
cpu	nv	512	2.3886358925E+04
gpu	nv	200	5.8388704911E+03
gpu	nv	300	1.0719400487E+04
gpu	nv	350	1.3505371797E+04
gpu	nv	450	1.9683902929E+04
gpu	nv	400	1.6498004800E+04

Table C.2: Correctness check, shows norm_rhs value for each run.

C.2. Plane sweep

hardware	com+op	size	copy in	sweep time	copy out	data	combined
cpu	nv	128	0	3.12E-01	0	0.00E+00	0.311542
cpu	ifx	128	0	3.37E-01	0	0.00E+00	0.3367
gpu	nv pinned	128	0.170941	1.03E-02	9.94E-02	2.70E-01	0.28071
gpu	nv	128	0.177871	1.02E-02	9.83E-03	1.88E-01	0.197947
cpu	gf	128	0	4.29E-01	0.00E+00	0.00E+00	0.429
gpu	nv pinned	256	0.223817	3.78E-02	7.59E-02	3.00E-01	0.337605
cpu	nv	256	0	3.25E+00	0	0.00E+00	3.248749
cpu	ifx	256	0	3.26E+00	0	0.00E+00	3.2561
gpu	nv	256	0.278018	3.80E-02	7.58E-02	3.54E-01	0.391757
cpu	gf	256	0	4.27E+00	0.00E+00	0.00E+00	4.27
gpu	nv pinned	2x256	0.232451	5.57E-02	0.149171	3.82E-01	0.43736
gpu	nv	2x256	0.200419	5.84E-02	0.149187	3.50E-01	0.408049
cpu	ifx	2x256	0	4.3412	0	0.00E+00	4.3412
cpu	nv	2x256	0	5.851106	0	0.00E+00	5.851106
gpu	nv pinned	4x128	0.162426	1.88E-02	3.81E-02	2.00E-01	0.219288
gpu	nv	4x128	0.194561	1.91E-02	3.74E-02	2.32E-01	0.251091
cpu	ifx	4x128	0	1.08E+00	0	0.00E+00	1.0776
cpu	gf	4x128	0	1.44E+00	0	0.00E+00	1.438
cpu	nv multic	4x128	0	1.49E+00	0	0.00E+00	1.4923
cpu	nv	4x128	0	1.92E+00	0	0.00E+00	1.920031
gpu	nv pinned	16x128	0.222078	4.09E-02	0.174796	3.97E-01	0.437756
gpu	nv	16x128	0.193751	4.08E-02	0.173182	3.67E-01	0.407743
cpu	nv	16x128	0	9.58E+00	0	0.00E+00	9.580076
cpu	ifx	16x128	0	5.09E+00	0	0.00E+00	5.0916
cpu	gf	16x128	0	1.16E+01	0.00E+00	0.00E+00	11.626
gpu	nv pinned	16x256	0.317989	3.18E-01	1.809132	2.13E+00	2.44511
gpu	nv	16x256	0.217488	3.20E-01	1.203614	1.42E+00	1.740859
cpu	nv	16x256	0	5.52E+01	0	0.00E+00	55.17828
gpu	nv pinned	20x256	0.216439	3.74E-01	1.654989	1.87E+00	2.245019
gpu	nv	20x256	0.225497	3.73E-01	1.55533	1.78E+00	2.154124
cpu	nv	20x256	0	8.70E+01	0	0.00E+00	86.98479
gpu	nv	64x128	0.17565	1.36E-01	0.614158	7.90E-01	0.925554
gpu	nv pinned	64x128	0.191116	1.48E-01	0.60203	7.93E-01	0.941307
cpu	nv	64x128	0	2.91E+01	0	0.00E+00	29.07904
cpu	nv	64x256	0	2.28E+02	0	0.00E+00	227.6636
gpu	sp nv pinned	128x128	0.206989	2.13E-01	0.604368	8.11E-01	1.024527
gpu	sp nv	128x128	0.188833	2.43E-01	0.607084	7.96E-01	1.03883
cpu	sp nv	128x128	0	5.32E+01	0	0.00E+00	53.22514
gpu	nv	128x128	0.179607	2.62E-01	1.203861	1.38E+00	1.645751
gpu	nv pinned	128x128	0.184965	2.62E-01	1.203827	1.39E+00	1.650939
cpu	nv	128x128	0	5.90E+01	0	0.00E+00	58.99802
cpu	sp nv	128x256	0	4.18E+02	0	0.00E+00	417.8268

Table C.3: Performance data of the Plane sweep algorithm. The first column from the right shows the time including data transfer and the data column shows the total data transfer time. The sweep time column shows the completion time without data transfers. The second column shows the compiler that is used, including special options.

hardware	com+op	size	norm_phi
cpu	nv	128	369.1216185
cpu	ifx	128	369.1216186
gpu	nv pinned	128	369.1216185
gpu	nv	128	369.1216185
cpu	gf	128	369.1216185
cpu	nv	256	746.6278093
cpu	ifx	256	746.6278093
gpu	nv pinned	256	746.6278093
gpu	nv	256	746.6278093
cpu	gf	256	746.6278093
gpu	nv pinned	2x256	1410.792634
gpu	nv	2x256	1410.792634
cpu	ifx	2x256	1410.792634
cpu	nv	2x256	1410.792634
gpu	nv pinned	4x128	1437.441846
gpu	nv	4x128	1437.441846
cpu	ifx	4x128	1437.441846
cpu	gf	4x128	1437.441846
cpu	nv multic	4x128	1391.020931
cpu	nv	4x128	1437.441846
gpu	nv pinned	16x128	5866.901269
gpu	nv	16x128	5866.901269
cpu	nv	16x128	5866.901269
cpu	ifx	16x128	5866.901269
cpu	gf	16x128	5866.901269
gpu	nv pinned	16x256	11863.58196
gpu	nv	16x256	11863.58196
cpu	nv	16x256	11863.58196
gpu	nv pinned	20x256	14850.0932
gpu	nv	20x256	1485.093202
cpu	nv	20x256	14850.0932
gpu	nv	64x128	23584.73896
gpu	nv pinned	64x128	23584.73896
cpu	nv	64x128	23584.73896
cpu	nv	64x256	47701.71681
gpu	sp nv pinned	128x128	47149.87
gpu	sp nv	128x128	47149.87
cpu	sp nv	128x128	47149.87
gpu	nv	128x128	47208.52246
gpu	nv pinned	128x128	47208.52255
cpu	nv	128x128	47208.52255
cpu	sp nv	128x256	94778.42

Table C.4: Correctness check, shows norm_phi value for each run.

Bibliography

Appelhans, D. (2023). Best practices for programming gpus using fortran, openacc, and cuda. https://www.nvidia.com/en-us/on-demand/session/gtcspring23-S51857/

Burlacu, T., Lathouwers, D., & Perkó, Z. (2023). A deterministic adjoint-based semi-analytical algorithm for fast response change computations in proton therapy. *Journal of Computational and Theoretical Transport*, *52*(1), 1–41. https://doi.org/10.1080/23324309.2023.2166077

Delftblue hardware. (2023). https://doc.dhpc.tudelft.nl/delftblue/DHPC-hardware/

Di Pietro, D. A., & Ern, A. (2011). *Mathematical aspects of discontinuous galerkin methods* (Vol. 69). Springer Science & Business Media.

Duderstadt, J. J., & Hamilton, L. J. (1976). *Nuclear reactor analysis*. Wiley.

The fortran programming language. (2022). https://fortran-lang.org/en/

The gnu fortran compiler 11.2.0. (2021). https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gfortran/

Kópházi, J., & Lathouwers, D. (2015). A space–angle dgfem approach for the boltzmann radiation transport equation with local angular refinement. *Journal of Computational Physics*, *297*, 637–668. https://doi.org/https://doi.org/10.1016/j.jcp.2015.05.031

Lathouwers, D. (2023). A deterministic approach for proton transport.

Levin, W., Kooy, H., Loeffler, J. S., & Delaney, T. F. (2005). Proton beam therapy. *British journal of Cancer*, *93*(8), 849–854.

Nvidia v100 tensor core gpu data sheet. (2020). https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf

Openacc api 2.7 reference guide. (2018). https://www.openacc.org/sites/default/files/inline-files/API%5C%20Guide%5C%202.7.pdf

Openacc programming and best practices guide. (2022). https://www.openacc.org/sites/default/files/inline-files/openacc-guide.pdf

Shi, Y. (1996). Reevaluating amdahl's law and gustafson's law.

Uilkema, S. B. (2012). Proton therapy planning using the S_N method with the fokker-planck approximation.

Witherden, F. D., & Jameson, A. (2020). Impact of number representation for high-order implicit large-eddy simulations. *AIAA Journal*, *58*(1), 184–197. https://doi.org/10.2514/1.J058434

Zheng-Ming, L., & Brahme, A. (1993). An overview of the transport theory of charged particles. *Radiation Physics and Chemistry*, *41*(4), 673–703. https://doi.org/https://doi.org/10.1016/0969-806X(93)90318-O