KTH Engineering Sciences

GPU Simulation of Rigid Fibers

ERIC WOLTER

Master's Thesis at School of Engineering Sciences
Supervisor: Katarina Gustavsson
Examiner: Michael Hanke

TRITA xxx yyyy-nn

Abstract

The major objective of this Master's thesis is to accelerate a serial implementation of a numerical algorithm for the simulation of slender fiber dynamics by using Graphical Processing Units (GPU). We focus on rigid fibers sedimenting due to gravity in a free-space Stokes flow. The ability to simulate a large number of fibers in a reasonable computational time on a high-performance parallel platform opens exciting new research opportunities.

The previous serial implementation is rewritten for parallel execution. The algorithm is implemented in single precision using the Compute Unified Device Architecture (CUDA) on NVIDIA GPUs. In addition, we develop an OpenMP version of the parallel implementation to run on multi-core CPUs. Using both implementations, we perform a number of benchmarks to determine the fastest variant of the algorithm. We observe a speedup of $20\times$ to $40\times$ on the NVIDIA GTX 970 compared to an Intel Core i7 4770. The GPU implementation can simulate up to 2000 fibers on a desktop computer and it takes only in the order of 8 seconds to advance one time step.

Furthermore, we have performed a number of simulations of known experiments for sedimenting fibers to validate the algorithm and to explore the numerical precision of the results. The results show an excellent agreement with results from prior experiments in the field.

Referat

GPU simulering av stela fibrer

Acknowledgements

Contents

List of Figures

List of Tables

List of listings

Chapter 1

Introduction

Predicting the physical behavior of particles suspended in fluid is of great interest in a variety of different field. The ability to accurately model different kind of materials allows their properties to be analyzed and optimized for a large number of applications. Examples include medical applications were the deliver and distribution of the active agent has to be modeled, waste management to efficiently extract waste from water and the paper industry which is trying to improve the properties of their material. Simulating these complex behaviors usually requires a large number of particles and being able to do so efficient is very important.

The work in this thesis focus on the numerical simulation on micro-scale. Rigid and fibers are allowed to sediment in a viscous fluid due to gravity. The fibers are simulated in a free-space Stokes flow. The model is expressed using a boundary integral formulation to simply and increase the performance of the simulation. This setup resembles the simplified model for paper pulp, where the quality of the paper is determined by individual fibers. By optimizing the fiber alignment and distribution different properties of the paper can be influenced. Accurately modeling the interaction requires many fibers to be simulated.

In this thesis we present a high performance GPU implementation of the rigid fiber suspension simulation. By taking advantage of the massively parallel architecture of modern GPUs many more fibers than before with a CPU based implementation can be simulated. We focus our work on the ability to easily and efficiently perform simulation on a high-end desktop computer or workstation readily accessible by the researcher. This approach allows the researcher to explore the huge problem space and simulate a relatively large number of fibers in a short amount of time. They can rapidly iterate to find and discover interesting test cases. Theses cases can than be used as a starting point for large scale

simulation using computing clusters.

We limit ourselves to the naive algorithm for simulating the fibers, were the interactions between every pair is computed. The development of such an naive algorithm is not only easier and more cost effective but the potentially large overhead introduced by a complex fast summation method, like multipoles might not result in desired performance increase. Additionally, because we are limited to a single precision simulation by our choice of consumer-grade GPU hardware, maintaining the accuracy of the simulation is a major concern. However, exploring theses questions is an interesting future research direction.

1. Prior research:
2. Two general approaches full 3d body simulation vs boundary integral formulation
3. Brief Examples/discussion for both
4. Parallel CPU/GPU research available for 3D Bodies?
5. Based on serial fortran implementation
6. OpenMPI version only interesting for computing clusters not desktop computers
7. Suffers from having to collect matrix to single node

This thesis into three major parts. First we introduce the theoretical foundation of the numerical method in chapter 2 and refer to the original paper by Tornberg and Gustavsson for in-depth details. We conclude the description of the previous work by discussing the numerical algorithm and the serial implementation on the CPU in chapter 3. In the following chapter 4 we briefly introduce the topic of general purpose computing on GPUs. Based on this we than describe our new parallel GPU implementation of the rigid fibers simulation in detail in chapter 5. The last part of this thesis presents the benchmarking result and achieved performance increase of the GPU implementation in chapter 6. Finally, we performed and compared our simulation results to a number of known experiments in chapter 7.

Chapter 2

Theoretical Foundation

The introduction discussed different applications of rigid fiber simulations. It especially stressed the importance of being able to simulate as many fibers as possible to generate the various patterns found in real world experiments.

In this chapter, we will present the theoretical foundation of the physics the simulations are based on. This is required to be able to understand the numerical method used throughout the rest of the thesis. However, the method is only presented in a reduced summary for more details please refer to [].

We will begin by introducing the Stokes flow and its fundamental solutions as they apply to suspended fibers. Afterwards, we will focus on the implications this has and how the flow properties can be calculated for the special case of straight and rigid fibers.

2.1 Stokes flow

The aim of the presented simulation is to model the behavior of objects immersed in a fluid. We restrict ourselves to rigid bodies, which are slowly sedimenting in a viscous fluid due to gravity. The velocity of these rigid objects is only dependent on their orientation and the resulting drag. The interaction between multiple objects is only modeled through the surrounding fluid as collision are completely excluded from the model.

$$\begin{aligned} \rho(\frac{\delta \mathbf{u}}{\delta t} + (\mathbf{u} \cdot \nabla)\mathbf{u}) &= -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned} \tag{2.1}$$

The Naviar-Stokes equations describe the motion of fluids and allow to model

the flow around an object. Using and solving theses equations is however quite challenging due to their time dependence and non-linearity. Fortunately, we can make a number of simplification due to the restriction we placed on our model. Our model is characterized by three properties.

1. *Newtonian fluid* — The viscosity μ of the fluid does not depend on the stress.
2. *Incompressible flow* — The density of the fluid does not change.
3. *Low Reynolds number* $Re = \frac{\rho U L}{\mu} \ll 1$ — The fluid velocities are very slow and/or the viscosity is very large.

Given these constraints and especially the very low Reynolds number due to slowly sedimenting objects in a viscous fluid the Naviar-Stokes equations can be linearized to arrive at the Stokes Equations

$$\begin{aligned} \nabla p - \mu \Delta \mathbf{u} &= \mathbf{f} \qquad \text{in} \quad \Omega, \\ \nabla \cdot \mathbf{u} &= 0 \qquad \text{in} \quad \Omega, \end{aligned} \tag{2.2}$$

where $\mathbf{u}(\mathbf{x})$ denotes the velocity field, $p(\mathbf{x})$ the pressure field and $\mathbf{f}(\mathbf{x})$ the force acting on the fluid at the location $\mathbf{x} = (x, y, z) \in \mathbb{R}^3$. The constant μ is the viscosity of the fluid.

2.2 Stokeslet

The Stokes Equations are linear in both the velocity and pressure, which allow them to be solved using a number of different methods for linear differential equations. Additionally, the equations are time independent and time dependence is only reintroduced due to the motion of the immersed objects and the boundary conditions. Thus given the boundary conditions the structure of the flow can be calculated.

For our model we define two boundary conditions. First we force the fluid velocity at the boundary of the object to be equal to the velocity of the object itself. These no-slip conditions on the surface of the objects are defined as

$$\mathbf{u} = \mathbf{u}_\Gamma \quad \text{on} \quad \Gamma, \tag{2.3}$$

where Γ denotes the union of all body surfaces and $\mathbf{u}_\Gamma$ the corresponding surface velocity, thus constraining the fluid to have zero velocity relative to the surface boundary.

The second boundary condition models the fact that the fluid velocity far from the object is not affected by its presence. This is modeled by the velocity field should be equal to a background velocity $\mathbf{U}_0(\mathbf{x})$ at infinity

$$\mathbf{u} \to \mathbf{U}_0 \quad \text{as} \quad \|\mathbf{u}\| \to \infty, \tag{2.4}$$

enforcing that the velocity field $\mathbf{u}$ should be equal to a background velocity $\mathbf{U}_0(\mathbf{x})$ at infinity. In our simulations this background velocity is always set to 0.

By taking advantage of the linearity of the Stokes equations and these boundary conditions so-called fundamental solution can be found. For this the force term in Stokes equation (2.2) is replaced by a point force acting at the origin $\mathbf{x}_0$

$$\begin{aligned} \nabla p - \mu \Delta \mathbf{u} &= \mathbf{F} \cdot \delta(\mathbf{x} - \mathbf{y}), \\ \nabla \cdot \mathbf{u}) &= 0, \end{aligned} \tag{2.5}$$

where $\delta(\mathbf{x} - \mathbf{y})$ is Dirac delta function. Solving these equations for the velocity field $\mathbf{u}(\mathbf{x})$ yields

$$\begin{aligned} u_i(\mathbf{x}) &= \frac{1}{8\pi\mu} S_{ij}(\mathbf{x}, \mathbf{y}) F_j \\ S_{ij}F_j &= S_{i1}F_1 + S_{i2}F_2 + S_{i3}F_3 \quad \text{where,} \\ S_{ij}(\mathbf{x} - \mathbf{y}) &= \frac{\delta_{ij}}{|\mathbf{x} - \mathbf{y}|} + \frac{}{|\mathbf{x} - \mathbf{y}|^3} \quad , \quad i, j = 1, 2, 3. \end{aligned} \tag{2.6}$$

The term S_{ij} is called the Stokeslet and is the fundamental solution for the Stokes equations. Figure 2.1 illustrates an example of the force $\mathbf{F}$ at the source point $\mathbf{x}_0$ and resulting velocity $\mathbf{u}$ at the observation point $\mathbf{x}$.

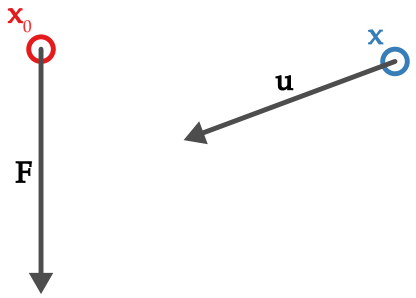


Figure 2.1: Singular point force embedded in stokes flow.

Additionally, we will later need higher order fundamental solutions which can be obtained by simply differentiating the Stokeslet, e.g. the so-called doublet is defined as

$$\mathbf{D}(\mathbf{r}) = \frac{1}{2}\Delta \mathbf{S}(\mathbf{r}) = \frac{1}{8\pi\mu}\left(\frac{\mathbf{I}}{|r|^3} - \frac{3rr}{|r|^5}\right). \tag{2.7}$$

2.3 Boundary integral formulation

Using the Stokeslet from equation (2.6) we can now model the influence of an immersed object on the Stokes flow. For a given point y as well as the the force distribution f on the surface of the object we can now calculate the resulting velocity at any point in the flow. The velocity u at the observation point x is then given by

$$u_i(\mathbf{x}) = \frac{1}{8\pi\mu} \int_\Omega S_{ij}(\mathbf{x}, \mathbf{y}) f_j(\mathbf{y})\, dS, \quad i = 1, 2, 3, \tag{2.8}$$

which is illustrated in Figure 2.2.

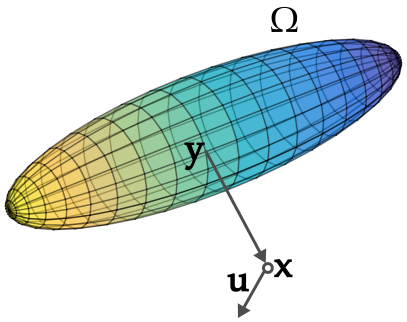


Figure 2.2: Immersed object in stokes flow.

Thus if f is known on the one hand the velocity field can be solved for any point x in the flow which is on the boundary or outside the object. If on the other hand only u is known on the boundary of the object we have to solve the boundary integral integration (2.8).

However for the simulation we don't know the velocity field u nor do we know the force distribution f on the surface of the object. The only force we know is the externally applied force, which in our case is simply gravity. Additionally we are only concerned with rigid objects. Thus by combining the equations governing rigid body motion with the no-slip boundary condition we can model the velocity field u on the surface Ω of the rigid body as

$$\mathbf{u}(\mathbf{x}) = \mathbf{U} + \omega \times (\mathbf{x} - \mathbf{x}_c) \tag{2.9}$$

where $\mathbf{x}_c$ and t denote the center point and orientation of the body and U and ω the translational and rotational velocity respectively. The rigid body motion is illustrated in Figure 2.3.

Furthermore we required that the integrated force distribution over the body as well as the torque have to be equal to the externally applied force. In our case

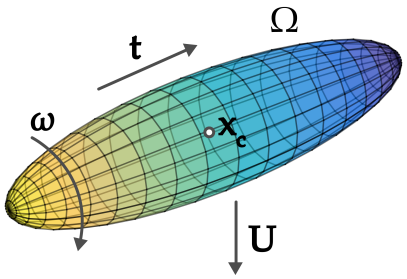


Figure 2.3: Immersed object in stokes flow.

this is simply gravity and an external torque of 0. For a single body the force $\mathbf{F}_{body}$ and torque $\mathbf{T}_{body}$ are thus defined as

$$\begin{aligned} \mathbf{F}_{body} &= \int_{\Omega} \mathbf{f}(y)\, dS_y \\ \mathbf{T}_{body} &= \int_{\Omega} (\mathbf{x} - \mathbf{x_c}) \times \mathbf{f}(y)\, dS_y \quad i,j = 1,2,3, \end{aligned} \tag{2.10}$$

and together with the equation relating the force and velocity for body

$$\mathbf{U}_i + (\omega(\mathbf{x} - \mathbf{x}_c))_i = \frac{1}{8\pi\mu} \int_{\Omega} S_{ij}(\mathbf{x}, \mathbf{y} f_i(\mathbf{y}))\, dS_y \tag{2.11}$$

we obtain the final boundary integral formulation. Until now we only looked at the model for a single body. However for the simulation we want to be able to handle many bodies. By taking advantage of the superposition principle resulting from the linearity of the Stokes equations, the equations (2.10) and (2.11) can easily be extended.

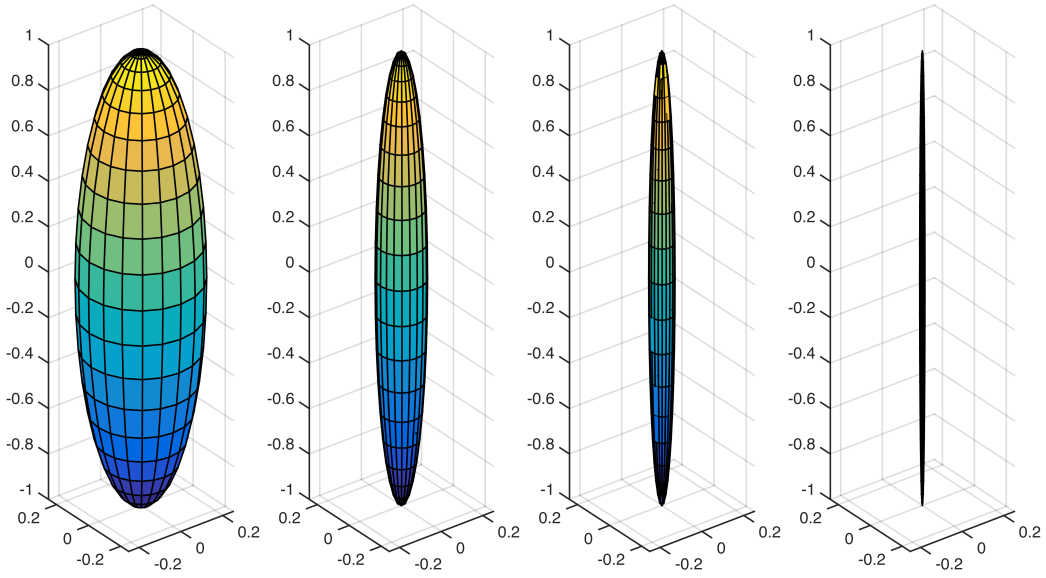
Reformulating the problem in form of a boundary integral reduced the original 3-dimensional partial differential equation to the 2-dimensional equation over the surface of the body. However for this many-body problem this is still expensive to solve. The next step to reduce the dimensionality of the model simplifies the rigid bodies to slender fibers.

2.4 Slender fibers

The goal of the simulation is to simulate many very slender fibers. For slender fibers the 2D-dimensional boundary integral formulation would be very expensive to solve numerically. We thus simplify the formulation of arbitrary rigid fibers by replacing them with ellipsoids, or more precisely in our case prolate spheroids.

They are defined by two parameters, $2 * L$ corresponds to the length of the major axis and a to the length of the minor axis or simply the fiber radius.

Additionally, we define the ratio $\epsilon = a/2L \ll 1$ and refer to it as the slenderness parameter. As illustrated in Figure 2.4, when ϵ becomes smaller the shape of the body asymptotically approaches that of an infinitesimal slender fiber.



(a) $\epsilon = 1/4$ (b) $\epsilon = 1/10$ (c) $\epsilon = 1/20$ (d) $\epsilon = 1/100$

Figure 2.4: Illustration slenderness parameters.

Without this approximation the fundamental solution have been evaluated on the surface of the rigid body. However, as [] showed the fluid velocity $\mathbf{U}(\mathbf{x})$ at any point x outside the fiber can be also be computed up to an accuracy of $O(\epsilon)$ by placing the fundamental solution on a 1-dimensional fiber centerline. The formulation remains closed for no-slip boundary condition and no angular variation in the fiber velocity.

Thus as shown in Figure 2.5 each fiber is only defined by its center coordinate and the orientation along its centerline parameterized by the arc-length $s \in [-L, L]$. In order to simplify the calculation the rigid fiber simulation uses a non-dimensional formulation of the numerical method. This allows each fiber m to be expressed in terms of

$$x_m(s,t) = x_m(t) + s * t_m(t) \quad \text{with} \quad s \in [-1, 1], \tag{2.12}$$

where x_m is the center coordinate and t_m the unit tangent vector representing the orientation.

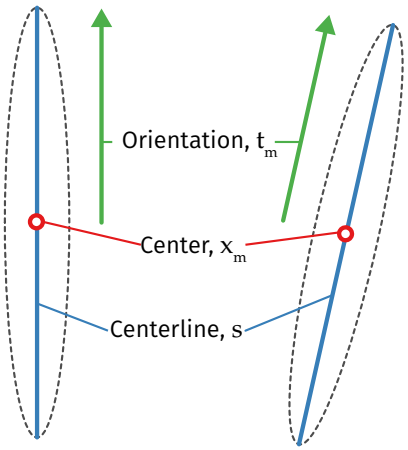


Figure 2.5: Slender fiber approximation.

By approximating the slender bodies using slender fibers we can reduce the dimensionality of the boundary formulation from 2D to a 1D integral equation. This allows for a much cheaper numerical simulation. The next step is to combine the equations (2.11) and (2.10) with non-dimensionalized rigid fiber model in equation (2.12) to arrive at the final mathematical model of the rigid fiber simulation.

For the slender fiber approximation the velocity for the centerline of fiber m is given by

$$\begin{aligned}\dot{\mathbf{x}}_m + s\dot{\mathbf{t}}_m = L(\mathbf{t}_m)\mathbf{f}_m(s) \quad &\textit{(Local contribution)} \\ + (\mathbf{I} + \mathbf{t}_m\mathbf{t}_m)\bar{\mathbf{K}}[\mathbf{f}_m](s) \quad &\textit{(Global contribution)} \\ + \sum_{\substack{l=1 \\ l\neq m}}^{M} \int_{-1}^{1} \mathbf{G}(\mathbf{R}(s,s'))\mathbf{f}_l(s')\,ds' \quad &\textit{(Contribution from other fibers)}\end{aligned} \tag{2.13}$$

where $\dot{\mathbf{x}}_m$ and $\dot{\mathbf{t}}_m$ denote the translational and rotational velocity of the fiber, respectively. In this equation we take advantage of the superposition principle in the Stokes flow and simply sum up the contribution over every other fiber l to the current fiber m. So for the contribution from another fiber l we integrate over the Stokeslet and Doublet over the complete fiber length for $s' \in [-1, 1]$. The function $\mathbf{G}$ than simply describes a linear combination of the two fundamental solutions

$$\mathbf{G}(\mathbf{R}) = S(\mathbf{R}) + \epsilon^2\mathbf{D}(\mathbf{R}). \tag{2.14}$$

By discretizing the integral with a quadrature rule we can calculate the equation (2.13) for a specific number of quadrature points along each fiber. In this case

the following function

$$\mathbf{R}(s, s') = \mathbf{x}_m + s\mathbf{t}_m - (\mathbf{x}_l + s'\mathbf{t}_l), \tag{2.15}$$

simply compute the difference vector between the two points on the fiber. Figure 2.6 illustrates the case of only two fibers with 5 quadrature points each.

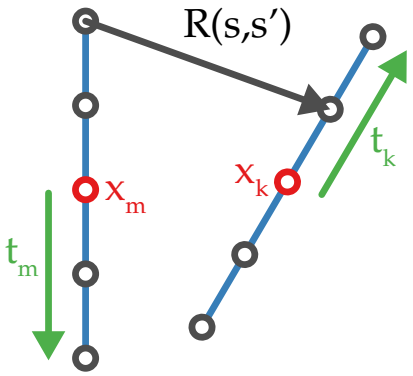


Figure 2.6: Contribution between fibers.

Right now we have the we have only one equation but three unknowns. The translational velocity $\dot{\mathbf{x}}_m$, the rotational velocity $\dot{\mathbf{t}}_m$ and the force $\mathbf{f}_m$ acting on the fibers. To close the system we again constrain the force and the torque for each fiber as

$$\begin{aligned} \mathbf{F}_m &= \int_{-1}^{1} \mathbf{f}(s)\, ds = \mathbf{F}_g \\ \mathbf{T}_m &= \int_{-1}^{1} s(\mathbf{t}_m \times \mathbf{f}_m(s))\, ds = 0. \end{aligned} \tag{2.16}$$

The total force integrated over each discretized quadrature point is always equal to the externally applied force, which as discussed before is just gravity. For the torque the same principle applies, however instead of an external influence we always force it to 0. Together with this the system is now closed and we can solve our rigid fiber simulation using the equations (2.13) and (2.16).

Chapter 3

Numerical algorithm and serial implementation

In the last chapter, we presented the theoretical foundation of the physics and mathematics involved in simulating rigid fibers. Based on the Stokes Equation, we introduced the framework to efficiently model the behavior of rigid fibers. Using this background we will now review the approach used for the numerical simulation.

We will separate the algorithm into four steps and discuss each individually. Additionally, we will touch upon implementation details used in the original serial version. We close the chapter with a brief reflection of the performance characteristics of the serial implementation to guide the parallel implementation on the GPU.

3.1 Assemble System

In the previous chapter we obtained the final closed system and the equations (2.13) and (2.16). By integrating equation (2.13) from -1 to 1 we can derive two separate equations for $\dot{\mathbf{x}}_m$ and $\dot{\mathbf{t}}_\mathbf{m}$. Using these two equations and some algebra we can obtain a system of equations for $\mathbf{f}_m$ for all fibers from $1, \dots, M$, which only include computable quantities. For more details and an in depth discussion please refer to the original paper by Tornberg and Gustavsson [14].

In order to be able to solve these equations for $\mathbf{f}_m$ we have to discretize them.

For this we expand the force as a sum of $N+1$ Legendre polynomials $P_n(s)$

$$\mathbf{f}_m = \frac{1}{2}\mathbf{F}_g + \sum_{n=1}^{N} a_m^n P_n(s), \tag{3.1}$$

where the coefficients a_m^n are vectors with three components. The number of Legendre polynomials N used for the force expansion is a parameter and is set to 5 in our simulation.

3.1.1 Closed linear system

Using the force expansion from equation 3.1 in equation 2.13 we get a closed linear system of equations for the coefficients a_m^n for $n = 1, \dots, N$ force expansions and $m = 1, \dots, M$ fibers.

For a 3-dimensional simulation, this results in a linear system of size $3MN \times 3MN$. The first step of the algorithm is thus to compute and assemble the linear system in memory. Writing the system in the standard form $\mathbf{A}\bar{\mathbf{a}} = \mathbf{b}$ gives the following structure for the matrix A

$$\mathbf{A} = \begin{bmatrix} \mathbf{I} & \bar{A}_{12} & \cdots & \bar{A}_{1M} \\ \bar{A}_{21} & \mathbf{I} & \cdots & \bar{A}_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{M1} & \bar{A}_{M2} & \cdots & \mathbf{I} \end{bmatrix}. \tag{3.2}$$

In this notation $\bar{A}_{ml}$ describes the $3N \times 3N$ matrix encapsulating the contribution from the force coefficients on fiber l onto the force coefficients for fiber m.

3.1.2 Inner integral

For each $\bar{A}_{ml}$ we need to compute the 3×3 matrix Θ_{lm}^{kn} from equation (**??**)

$$\Theta_{lm}^{kn} = \int_{-1}^{1} \left[\int_{-1}^{1} \mathbf{G}(\mathbf{R}(s, s')) P_k(s')\, ds' \right] P_n(s)\, ds, \tag{3.3}$$

for each force index $k, n = 1, 2, \dots, N$ which integrate the Legendre polynomials along each fiber. G and R are defined as in equation (2.14) and (2.15), respectively. Both the inner and the outer integral have to be evaluated for each fiber pair. This can be achieved by using a standard gaussian quadrature approach for both the inner and outer integral. However, as fibers get very close the number of

quadrature points have to increase to accurately compute the fiber interactions. Unfortunately, doing so also increases the unknowns in the system, which in turn decreases the performance. For our simulation we use the same approach as the original paper. We divide each fiber into 8 subintervals and use a three-point gaussian quadrature on each interval. This results in a total of $3 \times 8 = 24$ quadrature points per fiber, which represents a good trade-off between accuracy and performance.

Another option is to find an analytical solution for the inner integral and only solve the outer integral numerically. We will not discuss the detailed derivation of the analytical solution, for an in-depth discussion please see [**.**] In theory this approach allows for perfect accuracy for the inner integral, however in practice this is limited by the numerical precision of the simulation. The obtained formulas are recursive and thus sensitive to round off errors. To minimize the accumulation of round off errors, the original serial implementation uses a trick and switches the direction of the recursion, depending on how far apart the fibers are. This improves the practical accuracy and did not have a negative effect on performance.

Choosing between both options requires a careful examination of the accuracy and performance trade-off and is depend on the simulation setup. For the original serial implementation the combined numerical and analytical approach for evaluation proved to be the fastest and was thus chosen as the default. We will later explore how is applies to the new parallel GPU implementation.

3.2 Solve System

After having assembled the linear system $\mathbf{A}\bar{\mathbf{a}} = \mathbf{b}$ the next step is solving it. This can be achieved using standard linear equation solvers. This step is treated as a black box by the simulation, simply plugging in the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ and getting back the solution vector $\bar{\mathbf{a}}$ representing the force coefficients.

The linear system can either be solved using a direct solver or an iterative method like GMRES. Unfortunately the matrix is neither symmetric nor sparse so we can't take advantage of specialized solvers for these cases. As long as the fibers are sufficiently far apart and the matrix is not ill-conditioned, GMRES is able to solve the system in less than 10 iterations. This is the reason why the original serial implementation uses GMRES by default. How the different solvers perform on the GPU will be compared in section 6.3.

3.3 Update Velocities

The force coefficients obtained by solving the linear system can now be used to calculate the remaining unknowns. By using the force expansion in the two separate equations for $\dot{\mathbf{x}}_m$ and $\dot{\mathbf{t}}_\mathbf{m}$. The required implementation is thus similar to the computations for the *Assemble System* step. The two integrals, again, can be evaluated either exclusively numerical or with the combined approach of solving the inner integral analytically.

3.4 Update Fibers

The final step takes care of advancing the fibers forward in time. The equations do not impose any strict stability restrictions, so an explicit time-stepping scheme can be used. We also use the same second order multi-step method used in the original paper. The update for the position of the center coordinate $\mathbf{x}_m$ is given by the following discretization in time

$$\frac{3\mathbf{x}_m^{i+1} - 4\mathbf{x}_m^i + \mathbf{x}_m^{i-1}}{2\Delta t} = (2\dot{\mathbf{x}}_m^i - \dot{\mathbf{x}}_m^{i-1}), \tag{3.4}$$

where the time step is denoted by Δt and superscripts denote the numerical approximation of $\mathbf{x}_m(t_i)$. In order to compute the next state this method requires both the previous and the current state. As there is no previous time step for t_0, the first step $\mathbf{x}_m^1$ is computed by a simply first order forward Euler method. The calculation for the orientation vector $\mathbf{t}_m$ is computed using the same discretization by replacing the translational velocity $\dot{\mathbf{x}}_m$ with the rotational velocity $\dot{\mathbf{t}}_m$. Additionally, we must renormalize the orientation vector so that it maintains its unit length.

At the end of this step the state of the fibers can optionally be written to an external file for post processing and visualization in other tools. After completing this step, the algorithm starts again from the top with the *Assemble System* step. This cycle repeats until a specified number of time steps have been executed.

3.5 Algorithm summary

The original paper implemented this algorithm using Fortran. The computation were all performed in double precision and executed using a single thread on the CPU. In summary the four steps of the algorithm are

1. Assemble System
2. Solve System
3. Update Velocities
4. Update Fibers

Empirical results show that the majority of the required computation time is spent on the 1. *Assemble System* and 3. *Update Velocities* steps. The time required for advancing the simulation state in step 4. *Update Fibers* is completely negligible. We will see later in chapter 6 that the same holds true for the new parallel implementation.

Since we won't be writing our own implementation of linear solvers on the GPU we have only limited influence on the performance of the 2. *Solve System* step. We instead treat it as a black box and rely on the efficiency of pre-existing libraries. The only control we have is the choice which library and solver to use. For direct solvers the computation time only depends on the number of unknowns and is approximately constant throughout the simulation. The performance of iterative solvers on the other hand is highly depend on the condition number of the matrix and thus unpredictable. In line with the original paper we will both test a direct solver and iterative solvers to get a better understanding of their respective performance behavior.

Consequently, the most important step to optimize is 1. *Assemble System*. Fortunately it is well suited for parallelization. The fibers can be partitioned naturally across the compute units, where each unit is responsible for a subset of fibers. The focus of the optimization will thus lie on the *Assemble System*. As the required computation for the 3. *Update Velocities* step is similar it will also indirectly benefit from any optimizations. Additionally, we will also look at how the two different options for solving the integral in equation 3.3 perform in the parallel environment. Therefore, the next goal is to parallelize each algorithm step on the GPU.

Chapter 4

GPU Programming

In the previous chapter the numerical algorithm and its serial implementation was presented. It discussed various implementation details which have to be considered to arrive at the most efficient implementation.

To further increase the efficiency modern GPUs offer a massively parallel architecture to accelerate many different applications. We will begin with a short introduction to general purpose computing on GPUs. Afterwards, different Application Programming Interfaces (API) for programming on GPUs are discussed regarding their advantages and disadvantages. The chosen API, CUDA, will be presented with its programming model at the end of this chapter.

4.1 General Purpose Computing on GPUs

In the beginning Graphics Processing Units were highly specialized pieces of hardware developed to exclusively improve the performance of real-time 3D graphics. However, in recent years GPUs have started to be able to run arbitrary code instead of being limited to graphics related computations. This allows them to achieve impressive performance increases across a wide range of different general purpose applications. A good overview of the evolution of GPU computing can be found in Owens et al. [11].

The deciding factor for the performance is how well the application can be parallelized to take advantage of the massively parallel architectures of GPUs. This has lead to potentially large performance advantages of GPUs over CPUs as illustrated in Figure 4.1. It shows the year over year increase in the theoretical floating point operations per second (FLOP/s). The FLOP/s number is calculated by combining information of the number of compute cores, core frequency and

memory bandwidth for both the CPU and GPU models. This does not necessarily translate to direct real world performance increases but tries to visualize the potential GPUs have.

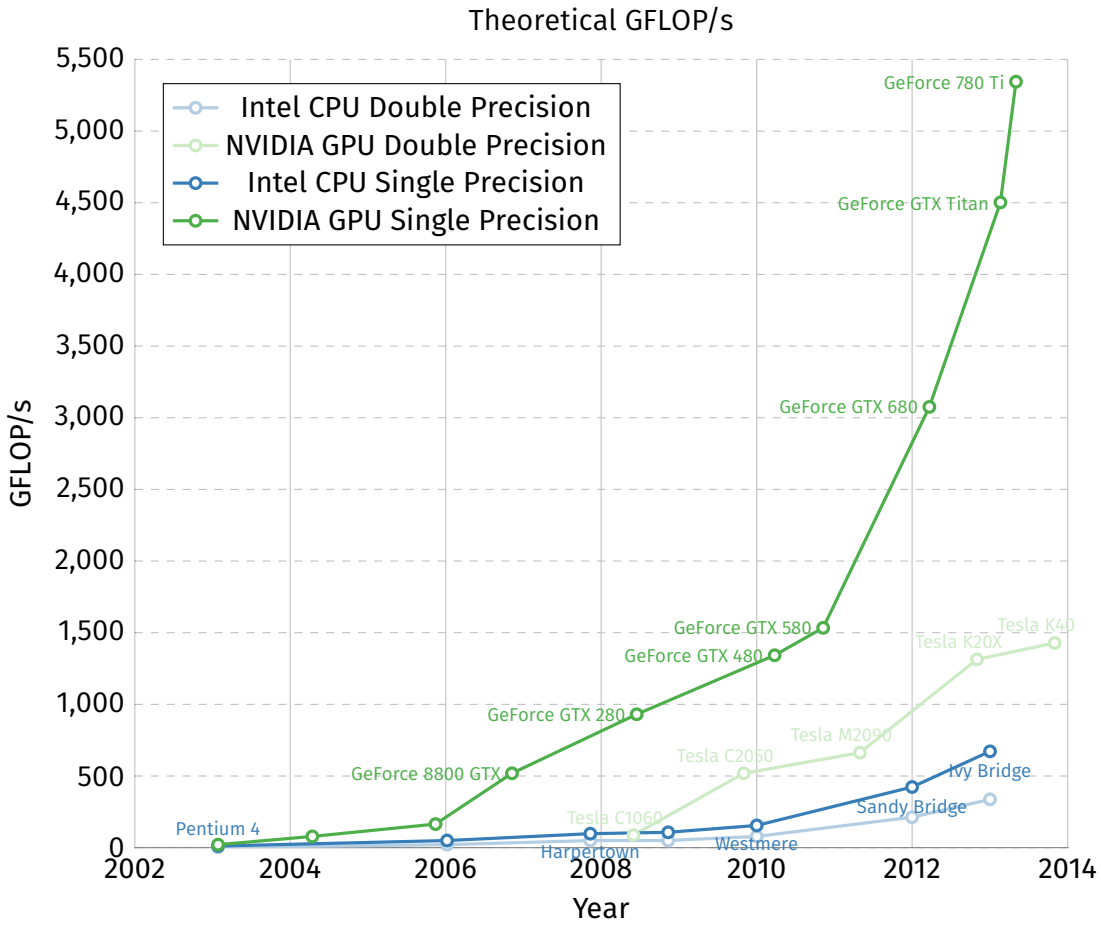


Figure 4.1: Year over year increase in theoretical floating-point operations per second for CPUs and GPUs[3].

The huge difference in performance mainly derives from the number of independent compute cores. Even though their individual cores are very fast, CPUs usually only have four, eight or maybe sixteen cores. In contrast to this, GPUs can have several hundreds of independent compute cores. Each core can simultaneously perform calculations and thus provides the opportunity to yield big performance improvements for high-throughput type computations. This fact also introduced general purpose computing on GPUs to the world of supercomputers. Over time, a growing number of them started supplementing their compute power

with GPUs and some even rely exclusively on GPUs for their computations.

In order to take advantage of these new massively parallel architectures new API had to be developed. The two proposed APIs are OpenCL and CUDA. OpenCL is an open and cross platform standard maintained by the Khronos Group. The same group is also responsible for its graphics focused counterpart OpenGL. OpenCL is not exclusive to GPUs, but instead tries to be a general abstract layer for different parallel architectures. This allows OpenCL code to be run not only on GPUs but also on CPUs. CUDA on the other hand is developed by NVIDIA exclusively for their line of GPUs.

4.2 CUDA vs. OpenCL

Choosing between OpenCL and CUDA is the first decision to be made when starting to implement a new project on GPUs. The main advantage of OpenCL is the ability to run on many different devices. All major players in the computing space provide an implementation on top of their platforms. Both Intel and AMD provide the API for their CPU and both AMD and NVIDIA have drivers available for their GPUs. However, this advantage can also be a disadvantage as the achievable performance might suffer from the abstraction across all platforms. The OpenCL framework is potentially not optimized for a particular device specific architecture. CUDA on the contrary is in theory highly optimized to achieve the best possible performance on NVIDIA's GPUs. In practice the difference can possibly be mitigated by spending the extra time to fine-tune the OpenCL implementation to the hardware's specific needs. Another disadvantage of OpenCL is the potentially outdated and inconsistent driver support for the various devices. This is especially true for NVIDIA who seem to have stopped updating OpenCL, still only supporting OpenCL 1.1 which was released back in 2010. Their main focus is on pushing CUDA and updating it to support all the feature in their new GPUs.

For this thesis we chose to go with NVIDIA's CUDA framework mainly because of the available hardware both at the workstation computers and at the local computing cluster. Additionally, this project does not need the cross-platform capability as the main focus is on pure performance in a highly specialized setup and simulation scenario. The application will not be widely distributed and only used for internal purposes.

4.3 CUDA Programming Model

The abbreviation CUDA stands for Compute Unified Device Architecture and was introduced by NVIDIA in 2006 as a general purpose parallel computing platform. It leverages the highly parallel architecture of modern NVIDIA GPUs to solve many different computational problems, which can lead to potentially large performance improvements compared to traditional CPUs.

The CUDA platform allows developers to use a variety of different options to program the GPU. The easiest way is to link to any CUDA-accelerated library and simply using the libraries interfaces from any software environment. For more advanced uses extensions to various programming languages exist like C/C++, Fortran and even managed languages like Java, Python and many more. This allows for easy and fast integration into any software environment the developer is comfortable with. Figure 4.2 illustrated the different components of the overall CUDA platform.

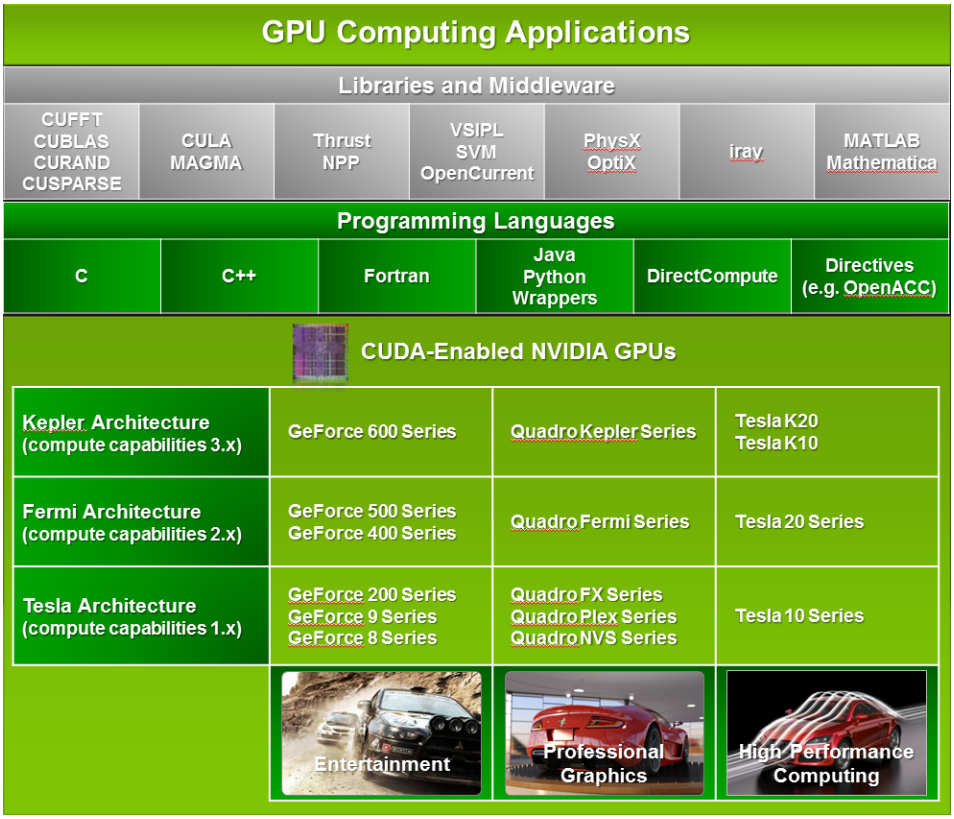


Figure 4.2: Overview of the CUDA platform[3].

The basic building blocks of the CUDA Programming Model from a development perspective are kernels. CUDA kernels are the equivalent of normal C functions. However, the major difference is that instead of being executed just once, kernels are executed in parallel by N different threads. These CUDA threads are distributed and run across the available compute cores of the GPU. To illustrate how a very basic kernel call looks, Figure 4.1 shows a code sample for a very simple vector addition.

```
// Kernel definition
__global__ void VecAdd(float *A, float *B, float *C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main()
{
  ...
  // Kernel invocation with N threads
  VecAdd<<<1,N>>>(A,B,C);
  ...
}
```

Listing 4.1: Pseudocode for CUDA vector addition.

CUDA Kernels It is important to remember that each kernel invocation is executed independently and no ordering is guaranteed. It is therefore essential to make sure to avoid any race conditions or shared memory access. There are however, ways to allow for shared memory access which will be briefly touched upon later in the practical implementation of the simulation.

Thread hierarchy In order to efficiently distribute the different threads across the compute cores of the GPU, CUDA defines a thread hierarchy. As discussed previously a GPU consists of many independent compute cores. On NVIDIA GPUs these cores are referred to as Streaming Multiprocessors (SMs). During execution of the application each SM is tasked with running a distinct set of threads. In CUDA these sets of threads are called thread blocks. Each thread block is then distributed to all the available SMs, which allows for automatic scalability depending on the number of SMs available as illustrated in Figure 4.3.

Thus the developer only has to divide the workload into appropriately sized

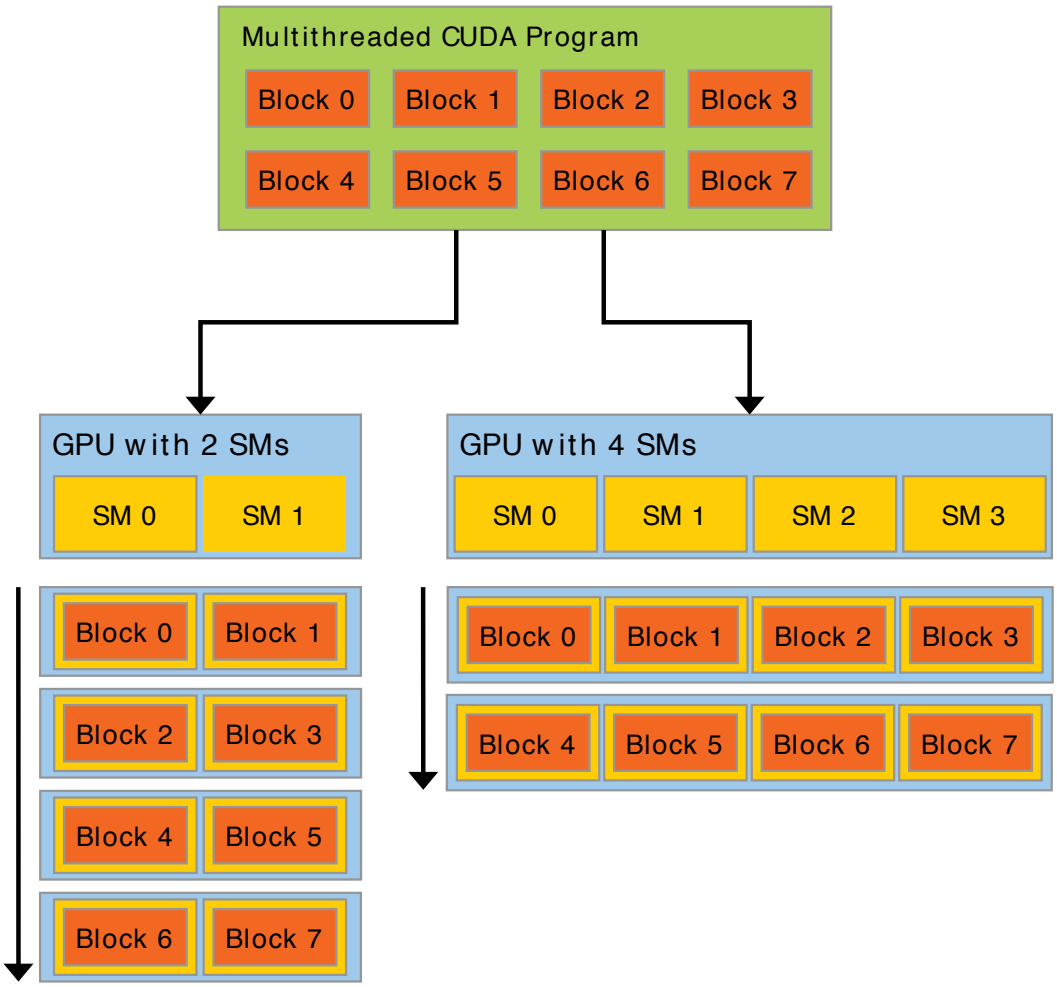


Figure 4.3: Automatic scaling of blocks across an arbitrary number of Streaming Multiprocessors[3].

blocks of threads and invoke the kernel. How to choose the optimal size of a block to maximize the performance is not an easy question to answer and is highly dependent on the particular task and implementation. In practice the size is often chosen by running benchmarks with various different sizes to determine the sweet spot.

In order to make programming and modeling of real world problems easier CUDA blocks can be addressed using either a one-dimensional, two-dimensional, or three-dimensional thread index. For example in the case of a matrix calculation it is more natural to think about parallelizing each element given by the row and column index instead of a single one-dimensional index. This is illustrated in the

code sample in Listing 4.2

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  ...
  // Kernel invocation with one block of N * N * 1 threads
  int numBlocks = 1;
  dim3 threadsPerBlock(N, N);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A,B,C);
  ...
}
```

Listing 4.2: Pseudocode for CUDA matrix addition, illustrating 2D thread blocks.

Finally, as the resources of each Streaming Multiprocessor are limited, there is an upper bound of how many threads a block can contain. Currently this maximum number of threads is 1024. This means that the maximum size of matrices possible to be added in the code sample in Listing 4.2 is 32×32. To solve this problem CUDA introduces another layer above blocks called a grid. A grid organizes thread blocks again into either one, two, or three dimensions. The number of thread blocks in a grid is unlimited and thus solely dependent on the size of the workload. Figure 4.4 shows an example configuration of a 2D grid with 2D blocks.

Memory hierarchy In addition to the Thread hierarchy CUDA also implements a 3 layer memory hierarchy as illustrated in Figure 4.5. Each level of this hierarchy differs in size, latency and scope. The fastest and smallest memory is private memory. Each thread has its own area of private memory and only it can read and write this area. How much space each thread has is determined by how many threads are executed on each SM. The more memory an individual thread requires the fewer threads can run in parallel on a SM. Optimizing this trade-off is important for achieving a good performance. On the newest GPUs each SM has $65536 * 32$-bit $= 256$KB to divide among the threads.

The second level is called shared memory, or sometimes confusingly local

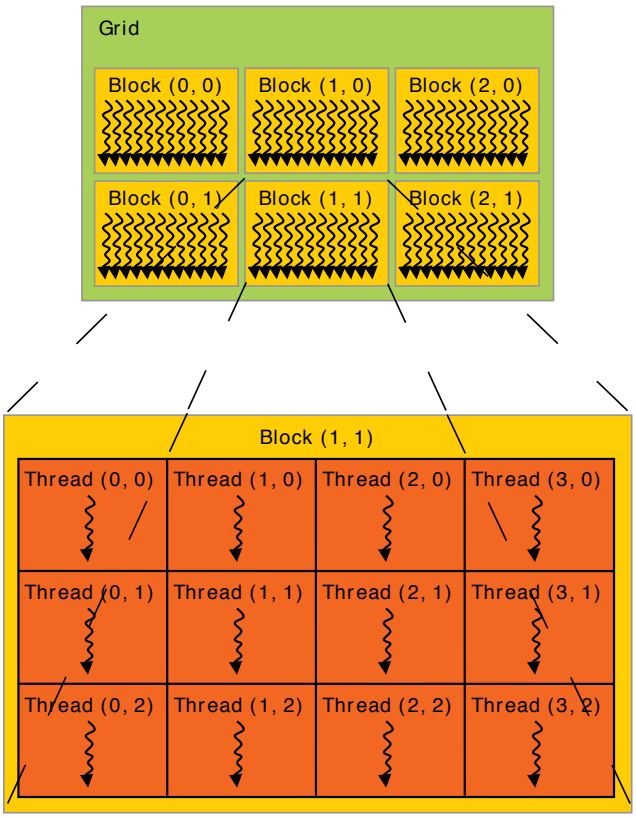


Figure 4.4: CUDA block addressing using 2D grid with 2D thread blocks.[3]

memory. Although it is slower than private memory, it has the advantage that all threads executing on an SM have simultaneous access to it. This allows different threads to cooperate and share work.

Depending on the characteristics of the algorithm, this can save a lot of computing time. Therefore it is important to analyze whether shared memory can actually improve the performance. The size of this memory level on current GPUs is around 64KB to 96KB.

The final and largest memory area is called global memory. This is the amount of memory used to advertise the GPU. Nowadays it is usually 2GB or 4GB. It is accessible by all threads and stores all data relating to the current application. However, accessing it can be quite slow compared to the other levels of the memory hierarchy. Additionally, one has to be careful to avoid memory conflicts when accessing the same memory location form different threads at the same time. This is especially important since this can further reduce the performance. Taking

everything into account, it can be said that it is an important optimization step to make access to global memory as efficient as possible. This can be achieved by intelligently packing data, or caching data in a lower level of the hierarchy. Another option to avoid the slow the performance can be used, if the required data for each individual thread from global memory is small enough. The associated latency when reading data from global memory can be hidden when enough calculation are performed on this data. The GPU is then able to quickly switch to another set of threads and continue the computation while waiting on data from global memory.

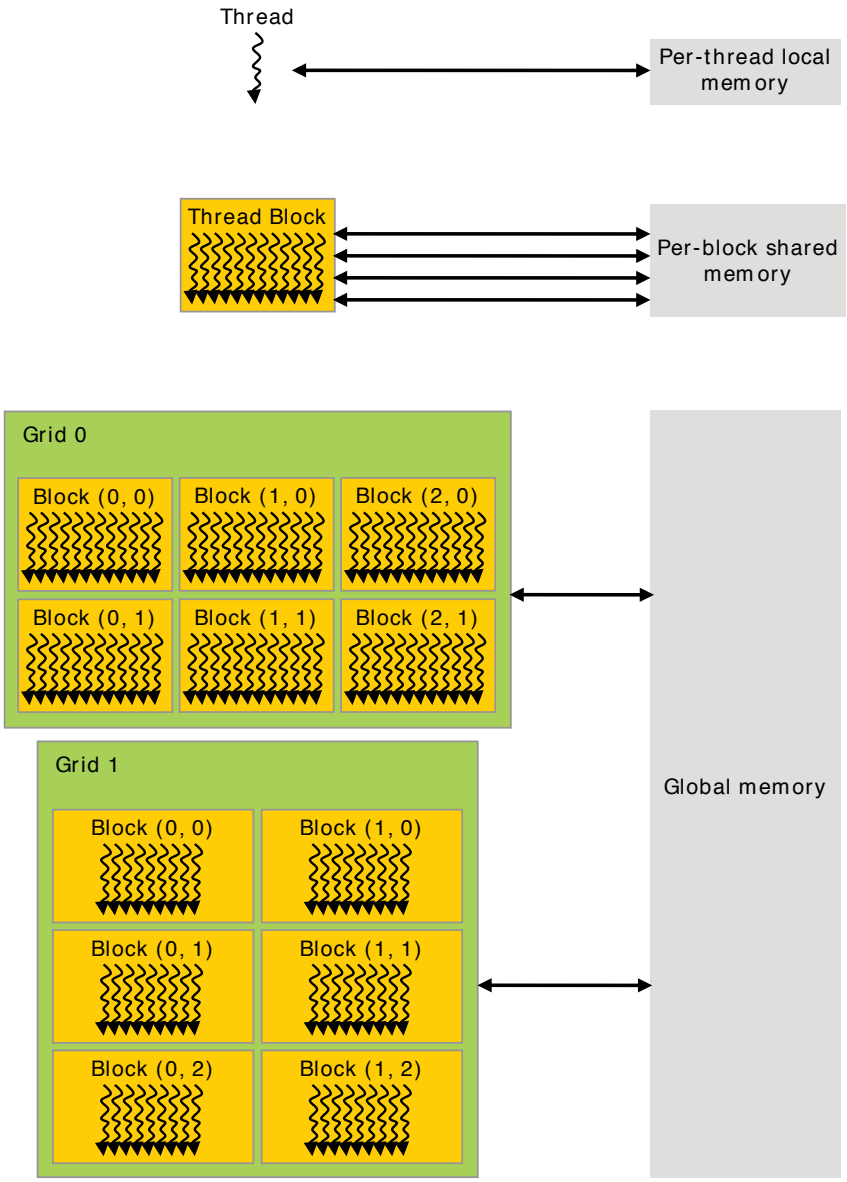


Figure 4.5: CUDA memory hierarchy[3].

Chapter 5

Parallel Implementation

Based on the previous two chapters describing the serial Fortran implementation and GPU programming we will now look at the algorithm in more detail and show, how it was adapted to take advantage of multi-core architectures. The main focus of this thesis is the implementation of the simulation on a modern massively parallel NVIDIA GPUs using CUDA. In order to gain a better understanding of the achievable performance improvements we additionally back-ported the finished GPU code of the algorithm to multi-core CPUs using the OpenMP framework.

We will begin by introducing the software tools and libraries utilized. Afterwards, we will explain the practical implementation of the algorithm in CUDA. This is followed by the discussion of several potential optimization approaches to further improve the performance of the simulation. The chapter ends with a brief explanation of OpenMP and how the code was parallelized on the CPU.

5.1 Development environment

The rigid fiber simulation developed as part of this thesis is only loosely based on the original serial Fortran implementation. This was done to ensure a clean starting point and avoid difficulties in adapting the existing code for parallel execution, as it was never intended to be run across multiple cores. This also provided the opportunity to learn from the shortcomings of the old code to not only parallelize it but also improve the efficiency in general.

The development was done exclusively on a Linux workstation running Ubuntu as this will also be the exact same runtime environment used in the later experimental usage of the resulting application. The build system for compiling and linking the final application was CMake. It was chosen because it is a widely used

open-source and cross-platform build system, which allows for easy integration of the various required libraries in a well documented and straightforward manner.

Under the hood the build system uses NVIDIA's CUDA platform tools to compile the code. For this NVIDIA includes *nvcc*, an LLVM-based CUDA compiler capable of compiling C/C++ code together with the CUDA specific extensions.

In order to facilitate easier usage of the application both during development and later real-world usage a Python wrapper script is also available. The script completely automates the building process and dynamically customizes the application code to support three different modes of operation. The first is a simple *run* mode which takes the supplied parameters and executes the simulation. The second mode is *validate*, it allows for a fully automated way to test and validate different algorithm variations against a known correct simulation run. This includes automatically computing the error as well as the error location in the matrix allowing for easier debugging of changes. The last mode is *benchmark* which runs the supplied parameters through a series of iterations collecting and aggregating timings for each simulation step as well as the total time.

5.2 Linear Solvers

In addition to the CUDA platform, the application also requires support libraries for the different linear solvers. It is very important to weigh all the options when choosing a particular linear solver algorithm or library, because solving the system can be one of the most time consuming steps. So the overall simulation time relies heavily on the ability to solve large and dense systems as fast as possible. The two main libraries are *MAGMA* for the direct solver and *ViennaCL* for the iterative solvers. Both will be introduced briefly now.

MAGMA / CuBLAS / OpenBLAS The MAGMA project contributes the implementation for the direct solver used in this thesis. This dense linear algebra library provides features similar to standard LAPACK functions but for multicore architectures. It also has features to support hybrid algorithms to run code across multiple GPUs or CPUs at the same time, however, these features were not explored in this thesis. Instead the focus was on a high performance single GPU implementation of a direct linear system solver.

MAGMA provides access to various high-level algebra routines, but the underlying math functions utilize the platform specific implementations of the BLAS levels. For CUDA this is implemented directly by NVIDIA in the form of the CuBLAS

libraries. Additionally, MAGMA tries to further improve their performance by combining GPU with CPU based algorithms. Thus a CPU based BLAS implementation is also needed. For this the OpenBLAS library was chosen which is the most up-to-date and high performance library available outside the very expensive Intel MKL library. OpenBLAS takes full advantage of multicore systems and is also used for the comparison of Fortran CPU implementation against the CUDA GPU implementation.

ViennaCL ViennaCL is an open-source linear algebra library developed at the University of Vienna. The library provides an abstraction layer across many different parallelization methods in order to facilitate consistent and easy to use support for BLAS level 1-3 and iterative solvers. This unique feature allows the developer to easily switch between different backends for parallelization. Currently, the library supports OpenMP, OpenCL and most importantly for this thesis - CUDA.

ViennaCL's focus is on solving sparse matrices with the implemented iterative solvers. However, it also has basic support for solving dense matrices using a variety of different iterative solvers. As the rigid fiber simulation exclusively relies on dense matrices this makes it an ideal candidate for benchmarking. For this thesis the BiCGStab as well as the GMRES iterative solvers were used and tested.

5.3 Kernels

The overall parallel algorithm is very similar to the serial version, however, each simulation step is separated into different kernels. Each kernel is invoked in a serial manner, this means CUDA guarantees that all data modified in a kernel is available before the next kernel is executed. These kernels are then distributed across the GPU. All calculations are done using single precision floating point numbers, as NVIDIA limits high performance double precision computation to their server class GPUs. The CUDA pseudocode for the algorithm is illustrated in Listing 5.1. It begins by parsing the simulation parameters and the initial fiber configuration. After that, the required memory is allocated on the GPU. Finally a simple loop executes all time steps and the four main algorithm sub steps — 1. *Assemble System*, 2. *Solve System*, 3. *Update Velocities* and 4. *Update Fibers* — are run on the GPU.

The application requires two general configuration files as an input. The first file is referred to as the parameters file which contains the different configuration variables and constants used throughout the algorithm. These include, for

```
int main()
{
  // Parsing algorithm parameters and initial fiber positions
  readParameters();
  readFiberConfiguration();
  allocateGPUMemory();
  ...

  for (int step = 0; step < max_timestep; step++)
  {
    AssembleSystem<<numBlocks, threadsPerBlock>>(...);
    SolveSystem<<numBlocks, threadsPerBlock>>(...);
    UpdateVelocities<<numBlocks, threadsPerBlock>>(...);
    UpdateFibers<<numBlocks, threadsPerBlock>>(...);
  }
  ...
}
```

Listing 5.1: Pseudocode for parallel algorithm on the host.

example the number and size of the time steps as well as the number of force expansion terms and quadrature points. Additionally, this file is also used to configure the iterative solvers like specifying the number of restarts for GMRES or the solution tolerance for BiCGStab and GMRES.

Each of the parallelized sub steps are now discussed in more detail. The purpose of each kernel as well as the required input and outputs.

Assemble System The *Assemble System* kernel is the most important step of the algorithm, as it is the most time consuming step. This was already discussed in section 3.5 and we will also be shown during benchmarking in chapter 6. The goal of this step is to build the matrix and vector in memory for the linear system of equations. As an example Listing 5.2 shows the pseudocode for the one-dimensional implementation of the *Assemble System* step. This means the code is parallelized for each fiber and each thread calculates the contributions to this fiber from all other fibers. Looking at the matrix in equation (3.2) each thread is thus responsible for $3 * N$ rows of the matrix, where N is the total number of terms in the force expansion. Alongside the one-dimensional implementation other options exist. They will be discussed in the optimization section 5.4.

The kernel requires two inputs, the current position of each fiber and its orientation. Using these combined with the equations outlined in chapter 2 and chapter 3 the matrix and vector elements are computed and used in the next step

```
__global__ void AssembleSystem1D(
  in float *positions,
  in float *orientations,
  out float *a_matrix,
  out float *b_vector)
{
  const int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i >= NUMBER_OF_FIBERS) return;

  for (int j = 0; j < NUMBER_OF_FIBERS ++j)
  {
    for (int force_index_j = 0;
         force_index_j < NUMBER_OF_TERMS_IN_FORCE_EXPANSION;
         ++force_index_j)
    {
      computeInnerIntegral(...);

      for (int force_index_i = 0;
           force_index_i < NUMBER_OF_TERMS_IN_FORCE_EXPANSION;
           ++force_index_i)
      {
        // Only 1D thread block
        // Each thread updates unique memory locations, thus
        // no need for atomics
        setMatrix(...)
        setVector(...)
      }
    }
  }
}
```

Listing 5.2: Pseudocode for the assemble system step with a 1D thread block.

to solve the linear system they define.

Solve System As this thesis does not aim to implement generic linear solvers, this step is treated as a black box. During the previous *Assemble System* kernel two arrays containing the matrix and right hand side of the linear system have been computed. These two arrays are now passed to the respective function of the library containing the linear solver. This is the MAGMA library in case of the direct solver and the ViennaCL library in case of the two tested iterative solvers BiCGStab and GMRES. Both libraries are able to directly use the already allocated memory regions and no additional allocations have to be performed. In order to conserve memory space the resulting solution vector is stored in the same

memory location as the b-vector and is passed on to the subsequent steps.

```
__global__ void UpdateVelocities2D(...)
{
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  const int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i >= NUMBER_OF_FIBERS) return;
  if (j >= NUMBER_OF_FIBERS) return;
  if (i==j) return;

  for (int quadrature_index_i = 0;
       quadrature_index_i < TOTAL_NUMBER_OF_QUADRATURE_POINTS;
       ++quadrature_index_i)
  {
    for (int quadrature_index_j = 0;
         quadrature_index_j < TOTAL_NUMBER_OF_QUADRATURE_POINTS;
         ++quadrature_index_j)
    {
      force = computeForce(coefficients, ...)
      computeDeltaVelocities(force)
    }
  }

  // 2D thread block
  // Each thread responsible for an interaction pair, thus
  // result is written to the same memory location
  // Using atomics to avoid conflicts
  atomicAdd(&(translational_velocities[i].x),
              delta_translational_velocity.x);
  atomicAdd(&(translational_velocities[i].y),
              delta_translational_velocity.y);
  atomicAdd(&(translational_velocities[i].z),
              delta_translational_velocity.z);

  atomicAdd(&(rotational_velocities[i].x),
              delta_rotational_velocity.x);
  atomicAdd(&(rotational_velocities[i].y),
              delta_rotational_velocity.y);
  atomicAdd(&(rotational_velocities[i].z),
              delta_rotational_velocity.z);
}
```

Listing 5.3: Pseudocode for the updating velocities simulation step.

Update Velocities After solving the linear system the solution coefficients are used to update the velocities of the fibers. The *Update Velocities* kernel accumulates the exerted forces for all fibers and updates both the translational and the rotational velocities simultaneously. In this particular instance the 2D thread

block version of the kernel is illustrated in Listing 5.3. This means each individual kernel invocation is responsible for a single pair of fiber interactions. In case two different threads calculate the interactions for the same fiber, it might happen that they try to write their results to the same memory location. When this happens, it could to undefined behavior and incorrect results for the velocities. Fortunately CUDA provides a mechanism to circumvent this issue.

By using the so-called atomic functions CUDA streamlines and serializes the memory access. Thus the *atomicAdd* function accumulates different velocity contributions to a fiber from all the other fibers. This ensures that each update to the memory location is handled in a serial manner, guaranteeing the correct value in memory. Of course this implies a potential performance degradation, however, newer GPUs with new CUDA versions have been very well optimized to only have a minimal impact. Benchmarking for the fibers simulation shows that using a 2D thread block and the associated performance increases far outweigh the potential performance hit of using atomics.

```
__global__ void UpdateFibers()
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i >= NUMBER_OF_FIBERS) return;

  next_positions[i] = 4/3 * current_positions[i]
    - 1/3 * previous_positions[i]
    + 2/3 * TIMESTEP
      * (2 * current_translational_velocities[i]
         - previous_translational_velocities[i]))

  next_orientations[i] = 4/3 * current_orientations[i]
    - 1/3 * previous_orientations[i]
    + 2/3 * TIMESTEP
      * (2 * current_rotational_velocities[i]
         - current_rotational_velocities[i]))

  normalize(next_orientations)
}
```

Listing 5.4: Pseudocode for the updating fibers simulation step.

Update Fibers The final simulation step takes care of advancing the position and orientation of the fibers in time. The pseudocode in Listing 5.4 implements the second-order multi-step method introduced in section 3.4. As it will be shown later

during benchmarking in chapter 6, the required time for this kernel is minuscule compared to the other steps. The kernel only scales linearly and in addition has a perfectly aligned memory access resulting in close to optimal usage of the GPU hardware.

5.4 Optimizations

During the development of the parallel GPU simulation great care was taken to continuously optimize the code both on an algorithmic level as well as on an implementation level. Numerous small code-level optimizations have been performed based on the original serial code like precomputing as much data as possible, avoiding variable allocations or unnecessary copy operations in performance critical sections of the code.

Additionally, more advanced optimization were made like rearranging calculations inside loops to avoid executing redundant calculations and consolidating multiple loops into one. Finally, techniques such as loop unrolling and faster math functions where also tested and included.

Throughout the optimization phase the benchmark suite was run after each step. This ensures that optimizations were only included if they had a measurable impact on the overall performance of the simulation. Moreover, with this approach potential performance regressions could be identified early and be avoided.

Many optimizations performed during this process are applicable to both the CPU and GPU, since they showed performance improvements for both. However, some optimizations and algorithm variations are uniquely suited to the GPU hardware. For this thesis we will look into three different optimizations on the GPU in more detail. The performance results for each will then later be discussed in chapter 6.

5.4.1 Numeric vs. Analytic Integration

The first optimization was already part of the original serial implementation as described in section 3.1.2. In the original paper [14] the authors observed that the analytical integration of the inner integral yielded a performance increase compared to the purely numerical integration. Generally an analytical integration should not only be preferred because of being faster but more importantly also because of being more accurate. However, for numerical precision reasons the actual implementation of the analytical integration can't achieve this theoretical

level of accuracy. Especially for configurations with fibers that are very far apart the recursive implementation suffers from round-off errors and numerical instabilities. The steps taken to minimize these instabilities potentially affect the performance.

Based on these considerations and the fact that the computation of the integrals is a very performance critical part of the implementation, exploring the performance implications of both approaches on the GPU is of great interest. In contrast to the serial CPU implementation of the original paper, our implementation on the GPU is actually faster when solving both integrals numerically. We will discuss the reason and consequences of this in more detail in section 6.2.1.

5.4.2 Shared Memory

As described in section 4.3 CUDA code is subject to a highly specialized memory hierarchy. Whereas traditional CPUs only have small caches and a large main memory pool, CUDA introduces the concept of a shared local memory space. The access time to this local memory is orders of magnitudes faster compared to accessing the global GPU memory. Additionally, local memory can be shared among the threads running on Streaming Multiprocessor and potentially save time by avoiding to constantly access the slow global memory.

In order to test this, shared memory was implemented and tested with the *Assemble System* step, the most performance critical kernel. To understand the idea, imagine the 2D thread block implementation of the kernel. Each thread block is responsible for many pairs of fiber interactions, e.g. fibers $[1, \dots, 8]$ each interacting with fibers $[9, \dots, 16]$. In total, these are $8 \times 8 = 64$ interactions. Each kernel invocation is responsible for one pair and has to load the position and orientation for the two interacting fibers. However, on closer inspection it is obvious that one does not need to load each fiber every time. As soon as fiber 1 has been loaded into shared memory it can be reused for all the interactions with fibers 9 through 16 avoiding the unnecessary and slow access to global memory.

How this affects performance is not always easy to tell, as various factors can influence the result. One factor, for example might be that the loaded data is small enough and repeated access to global memory can be automatically avoided by taking advantage of other memory caches. Another factor is performance characteristic of the kernel. Optimizing for shared memory usage only makes sense if the kernel is memory bound, meaning that most of the execution time is spent waiting for data. If on the other hand the kernel is compute bound the GPU is able use time effectively by performing pending computations while waiting for

memory access. In this case forcing threads in a block to wait for shared data can actually decrease the overall performance of the kernel.

However, in general efficient exploitation of shared memory can be a huge advantage for parallel GPU implementations. This is especially true when comparing the performance to CPUs, as they don't have an equivalent fast and comparatively large memory space. Unfortunately, during our testing in section 6.2.2 the effect for the GPU simulation of rigid fibers proved to be rather limited.

5.4.3 Thread Block Dimension

There are many different factors that determine the performance of a particular GPU algorithm. This is especially true with regard to optimally taking advantage of the specific underlying GPU architecture, which change even between different models of graphics cards. How to best utilize the hardware depends on specific memory access patterns, avoiding too much register usage and choosing optimal settings for the thread block size. Each graphics cards can have a different number of streaming multiprocessors with only limited resources and taking advantage of this can result in performance increases.

For this thesis we looked at the Thread Block dimension in particular and how choosing a different approach of parallelizing affects the performance. Doing so, the focus was on the *Assemble System* step, which is the most performance critical step. However, the results were then also transferred to the *Update Velocities* step.

The most straight forward approach is to simply parallelize the algorithm with regard to a single fiber. This means each kernel invocation is responsible for calculating all the interactions for this fibers with all other fibers. In this way a single kernel is responsible for multiple rows of the resulting linear system matrix. Additionally, this approach does not have any memory access conflict as each kernel only writes to the memory location belonging to its unique fiber. The potential disadvantage for a one-dimensional thread block, however, is that the resulting code can be more resource intensive for each single kernel and potentially hinder the performance on each multiprocessor.

For a two-dimensional thread block each kernel invocation is responsible for a pair of interacting fibers. While this decreases the necessary resources, it also requires atomics which can potentially slow down the execution. Three-dimensional thread blocks are the maximum allowed dimensions for a CUDA thread block. They are a further extension of the two-dimensional thread block, as now each kernel

invocation is not responsible for the complete interaction but only the interaction resulting from a specific point of the force expansion. This results in even more potential memory conflicts and also increases the total number of thread blocks which have to be distributed.

The kernel invocations for the one-dimensional and two-dimensional thread block approach are visualized using the matrix structure in equation (3.2), as shown in Figure 5.1. In the one-dimensional case each invocation is responsible for all sub matrices $\bar{A}_{ml}$ describing the contribution from the force coefficients on fiber l on to the force coefficients for fiber m. Thus in total this results in M kernel invocations, one for each fiber. For the two-dimensional case each invocation is only responsible for a single sub matrix. Here the total number of invocations is $M \times M$. The three-dimensional case then divides the calculation for the sub matrix further, one additional invocation per force index for a total of $M \times M \times N$ invocations.

$$\mathbf{A} = \begin{bmatrix} \mathbf{I} & \bar{A}_{12} & \cdots & \bar{A}_{1M} \\ \bar{A}_{21} & \mathbf{I} & \cdots & \bar{A}_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{M1} & \bar{A}_{M2} & \cdots & \mathbf{I} \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} \mathbf{I} & \bar{A}_{12} & \cdots & \bar{A}_{1M} \\ \bar{A}_{21} & \mathbf{I} & \cdots & \bar{A}_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{M1} & \bar{A}_{M2} & \cdots & \mathbf{I} \end{bmatrix}$$

(a) 1D (b) 2D

Figure 5.1: Illustration of 1D and 2D thread block dimension using simulation system matrix.

It is not clear, how exactly the performance is affected by each decision for the thread block dimension. Only trial-and-error benchmarking combined with metrics from CUDA can find the optimal setting for the specific algorithm. We will later show in section 5.4.3, that for our GPU implementation the two-dimensional approach is the most efficient one.

5.5 OpenMP

The goal of this thesis is to implement a high performance rigid fiber simulation on the GPU using CUDA. In order to better understand to what degree this goal was achieved, it is crucial to have a comparison. The original serial implementation is

not an ideal candidate as it does differ in a number of ways. First of all, it is purely serial and doesn't take advantage of todays multicore CPUs. Furthermore, it was implemented in double precision and because of arbitrary restrictions NVIDIA places on its consumer GPUs, they are only suited for single precision. That is why the CUDA implementation is strictly single precision. Finally, the primary focus of the original Fortran implementation was a correctly implemented algorithm and not performance.

For these reasons and in order to have a fairer comparison of the performance differences between the GPU and CPU implementation, a completely new and rewritten parallel CPU simulation was also implemented. For the parallelization on the CPU the OpenMP library was chosen.

After having implemented a parallel algorithm for the GPU, the conversion to the OpenMP-based CPU implementation was relatively straightforward. All optimizations done for the GPU implementation were also applied to the new CPU code when applicable. In order to parallelize the BLAS functions required for the linear solver the already included OpenBLAS library was chosen. OpenBLAS is an open-source and highly optimized library and automatically parallelizes BLAS functions using pthreads across all available CPU cores. In contrast to the GPU implementation, the OpenMP version is only parallelized in one-dimension. This means that each core calculates the interactions for one fiber with all other fibers or put differently all matrix rows belong to one fiber. As the underlying number of independent threads is much lower on CPUs, different parallelization dimensions didn't have an impact during testing.

The end results of the practical implementation for this thesis is a highly optimized CUDA implementation for NVIDIA GPUs and additionally a parallelized and optimized Fortran OpenMP implementation for CPUs. The next chapter will now look at a number of performance metrics and compare them between the GPU and CPU.

Chapter 6

Benchmarks

The last chapter introduced the parallel implementation of the numerical simulation. It contained a practical overview of the implementation of the algorithm using NVIDIA's CUDA framework. Additionally, possible optimizations to take advantage of the unique properties of the GPU architecture were outlined.

Using all the available implementations of the algorithm this chapter will showcase a multitude of different results and benchmarks performed. This is done to illustrate the achieved performance increases on the GPU over the original serial CPU implementation and the parallel OpenMP implementation.

6.1 Methodology

The methodology used for the benchmark suite is the same for all presented benchmarks. This ensures comparable results and fairest comparison possible. To give a brief overview of the hardware and benchmark scheme used, both are described in the next sections. This allows to better understand the conditions for the benchmarks and thereby put them in the right context.

6.1.1 Hardware

All benchmarks were run on the same workstation with specifications listed in Table 6.1. The components were chosen to provide a balanced system. This was done to come as close as possible to a fair comparison between the CPU and GPU, as comparing a high-end CPU against a low-end GPU would only be of limited value.

The Intel Core i7 4770 processor is an 4-Core CPU based on Intel's Haswell

Workstation	
Processor	Intel Core i7 4770
Graphics	NVIDIA GTX 970 4GB
RAM	16GB DDR3
Operating System	Ubuntu Linux 12.04 LTS
CUDA Driver	CUDA 6.5.16

Table 6.1: Benchmark Hardware Specification.

Architecture. With its 8 parallel threads and 3.4 GHz it was one of the top-of-line processor from 2013/14 and is currently still available for around $300\$$. The NVIDIA GTX 970 4GB is part of NVIDIA newest lineup of graphics cards based on the Maxwell Architecture. The main advantage of these new cards is the large memory of 4 GB allowing for larger simulations. With a current price of slightly above $300\$$ it fills the middle price class for all Maxwell cards. Overall this can be considered to be a balanced system.

6.1.2 Benchmark scheme

The main goal of the benchmark system was to generate statistical significant and reproducible performance numbers. To ensure this, all benchmarks for both the GPU and the CPU were run using the exact same scheme like the number of iterations and stop criteria. In order to obtain the timings, the built-in CUDA timing events were used for each individual kernel. For Fortran the *SYSTEM_CLOCK* function was used.

For all different benchmarks timings were obtained for a different number of fibers starting from 100 up to 2000 in 100 increments. For each separate number of fibers a number of iterations are run to obtain the average over multiple runs. For each iteration a completely new and random initial fiber configuration is generated with the current number of fibers. In order to exclude illegal configurations where fibers overlap and intersect an additional correction pass is done over the fibers to ensure a minimal and average distance between all fibers. For the configurations in this thesis the minimal distance was always set to 0.2 and the average to 0.4.

Using this semi-random generation the rigid fiber simulation is run for exactly 10 time steps. To avoid remaining outliers in the configuration potentially causing variations in the timings the first time step is excluded. The first time step is thus used as a simple warmup step for the simulation. So the final average time for

each run is taken from the last 9 time steps.

In order to ensure a statistical significant result, the number of iterations is not fixed. Instead it is determined dynamically and based on the relative standard error of the already collected timings for the particular number of fibers. A minimum number of iterations is always executed first and their relative standard error of the total times is calculated. If the relative standard error of the dataset is larger than a specified threshold, the number of iterations is doubled and run again.

```
for(int N = 100; N <= 2000; N += 100)
{
  array timings;

  int iterations = 4;
  while (iterations <= MAX_ITER)
  {
    for(int i = 0; i < iterations; ++i)
    {
      generateRandomInitialFiberConfiguration();
      run(10); // execute 10 timesteps
      timings.add(getTiming());
    }

    rse = calculateRelativeStandardError();

    if (rse <= 0.2)
    {
      break;
    }

    iterations = timings.count();
  }

  reportTimings();
}
```

Listing 6.1: Pseudocode for benchmark scheme.

The benchmark suite uses a minimum of 4 iterations to obtain the initial timings. In case the relative standard error (RSE) is not below a threshold of 20% an additional set of timings is measured. For the new run the number of iterations equals the number of already obtained timings, which means that the total number of timings is doubled in each round. This process repeats until the threshold is not exceeded and more reliable benchmark timings have been obtained. This algorithm for collecting the benchmark results is illustrated using

pseudocode in Listing 6.1.

6.2 Optimizations

We now look at the performance results for the different optimizations previously outlined in section **??**. Where applicable, the results will be compared between the OpenMP and the CUDA version of the algorithm.

6.2.1 Numeric vs. Analytic Integration

The first benchmark tests the performance of the two different approaches to compute the inner integral from equation (3.3). It can be solved either numerically or analytically. Figure 6.1 illustrates the performance timings for the *Assemble System* step of the parallel OpenMP version. Inline with the observations made by the authors of the original serial implementation [14] the parallel version of the analytic integration is always faster than the numeric integration.

In contrast to the expected results of the OpenMP implementation the CUDA implementation shows a very different picture. When we look at the graph for the CUDA implementation in Figure 6.2, we can observe that the results are reversed. The numerical integration outperforms the analytical integration by a large margin increasing with the number of fibers.

The reason for this result lies in the scheduling and execution of work on the GPU. All code inside a thread block (more precisely a warp) is always executed in lockstep. This means each line of code is executed for each thread in parallel. However, if the code encounters a branch in the execution path, like a simple *if* statement, the threads diverge. First, all threads for which the condition is true are executed while the other threads have to wait. Then all threads for which the condition is false are executed while the others are not used. Finally, after all divergent code paths have been executed the code continuous in lockstep. This issue is referred to as Branch Divergence and should be avoided as much as possible when writing parallel GPU Code.[2]

To confirm that Branch Divergence is the reason for the slowdown of the analytic integration on the GPU we look at the metrics of the CUDA profiler *nvprof*. The metric *Warp Execution Efficency* shows the ratio of the average active threads per warp to the maximum number of threads per warp. The metrics for both the numerical and analytical integration of the Spherical fiber setup used in section 7.3 can be seen in Table 6.2.

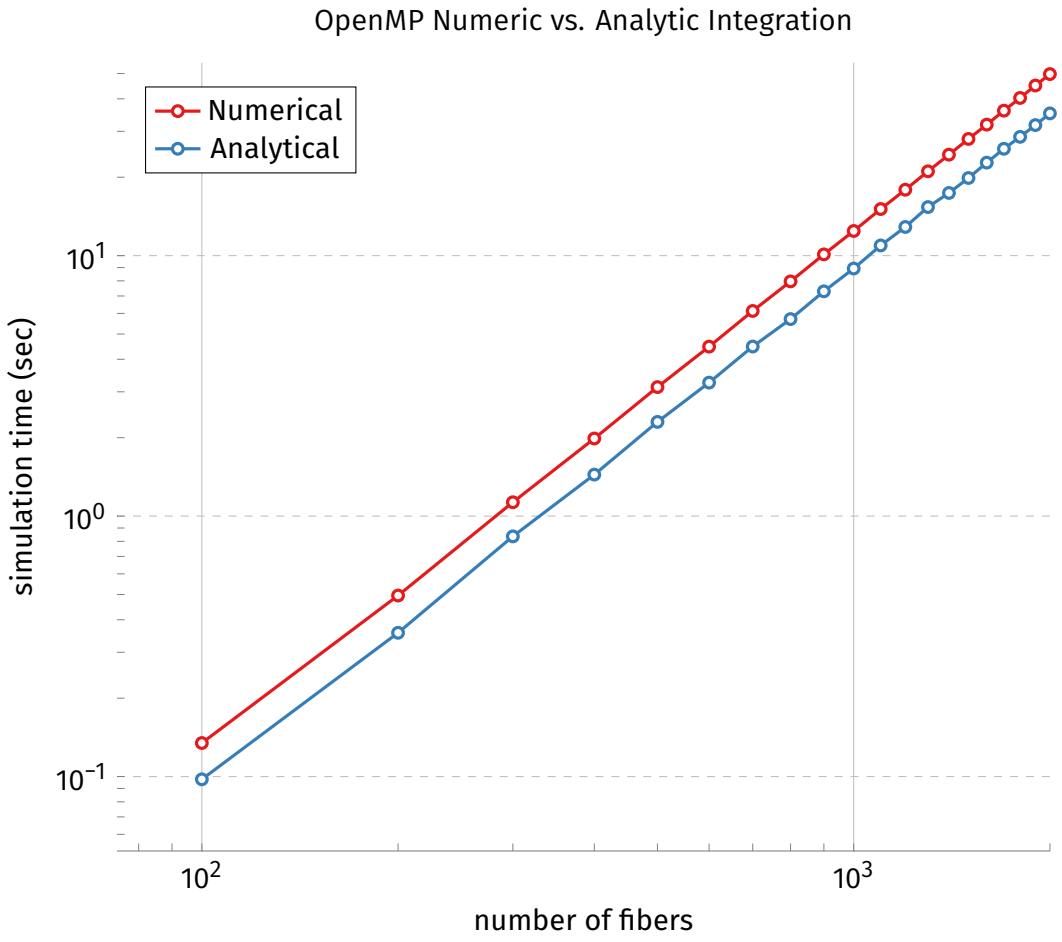


Figure 6.1: Benchmark computing integrals using OpenMP.

Algorithm	warp_execution_efficiency
Numerical	99.01%
Analytical	53.79%

Table 6.2: Warp Exection Efficiency of Numerical vs. Analytical Integration.

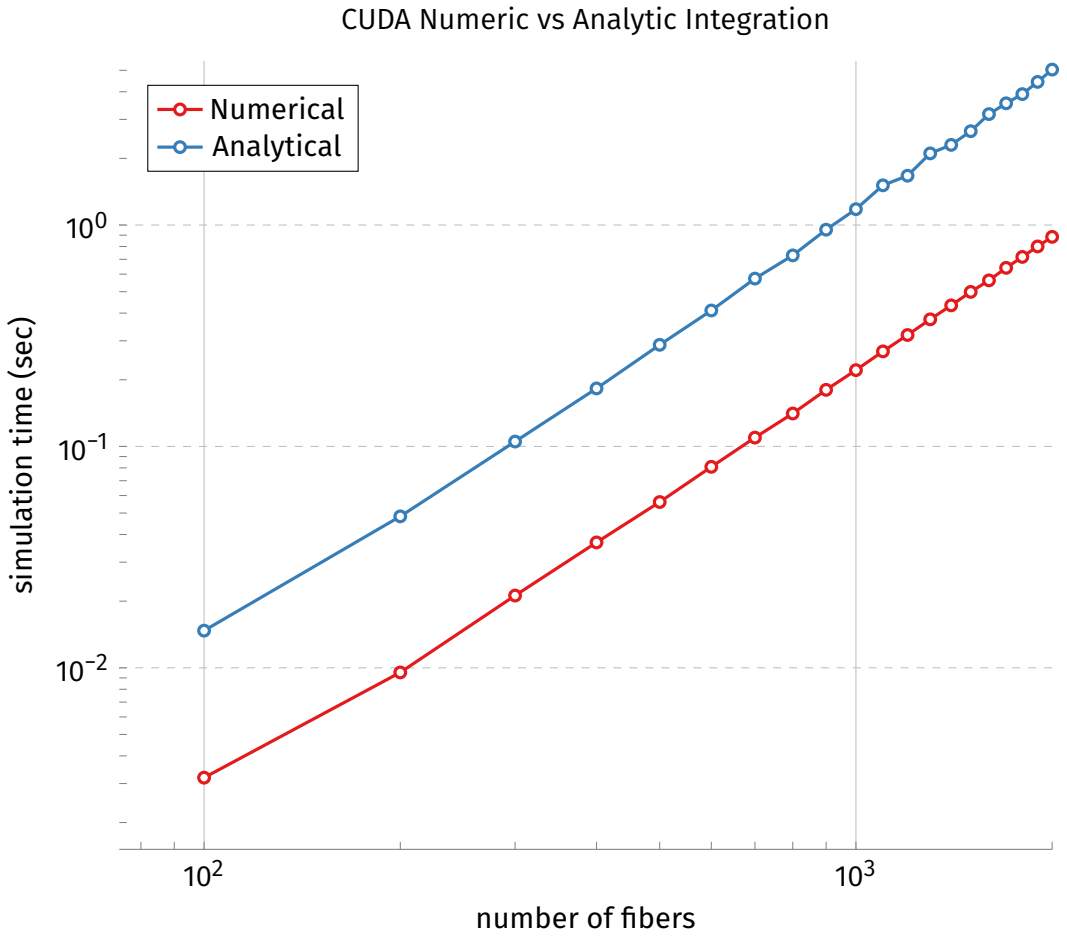


Figure 6.2: Benchmark computing integrals using CUDA.

On the one hand the numerical integration is almost 100% efficient, meaning all warps execute in complete lockstep. The analytical integration on the other hand is only 50% efficient, meaning that most of the time only half of the threads actually perform work while the other half is just waiting. This results in the observed performance difference. As already discussed in section 5.4.1 the analytical solution of the inner integral potentially suffers from numerical instabilities. Closer inspection of the source code reveals that the steps taken to minimize these instabilities are responsible for the Branch Divergence. The steps involve a simple *if* statement, that switches between two code path depending on how far apart the two fibers are. Unfortunately, this workaround is unavoidable to ensure numeric stability and explains the decreased performance on the GPU.

6.2.2 Shared Memory

The second optimization tried to use shared memory to reduce the amount of data that had to be transferred between the compute units and the global memory. For this each Streaming Multiprocessor has a small amount of locally shared memory. This memory can be accessed from all threads on the SM and in case data can be shared, it only has to be transferred slowly from global to shared memory once. Afterwards, in can be accessed from the faster local memory.

During testing and benchmarking the shared memory implementation of the *Assemble System* step as described in section 5.4.2, no performance effect could be observed. Even though data can theoretically be shared among different threads is does not result in saved execution time.

The reason for this is that the *Assemble System* step is compute bound and not memory bound. This means that the time it takes to execute the computations, e.g. solving the integrals, takes substantially longer than reading and writing to global memory. This can be explained by looking at the 2D thread block version, where each kernel is responsible for a pair of fibers. Each kernel only needs to read 4 vectors from global memory, the position and orientation of both fibers. Assuming single precision this is a total of just $4 * 3 * 4$bytes $= 48$bytes per kernel invocation.

While waiting for this data from global memory, CUDA is able to quickly switch between different sets of threads and continue the computation there. Thus the only waiting time occurs when the very first set of data has to be loaded. After the first data has arrived computations can be performed on it. While these longer running computations are happening, other threads can already issue data loading requests. Once the first computation finishes and the next set of threads is executed the data has already been fetched. As there was no advantage in using shared memory in the compute bound *Assemble System* step, we opted for the simpler implementation without it for all our simulations.

6.2.3 Thread Block Dimension

The next optimization looked at was the Thread Block Dimension on the GPU. Choosing the best option is a trade-off between the resources used and the overhead caused by an increased amount of memory writes to the same location. Writing to the same memory location from different threads would result in undefined behavior and avoiding this requires potentially slow atomic functions.

The results in Figure 6.3 indicate that the best option for this particular GPU is

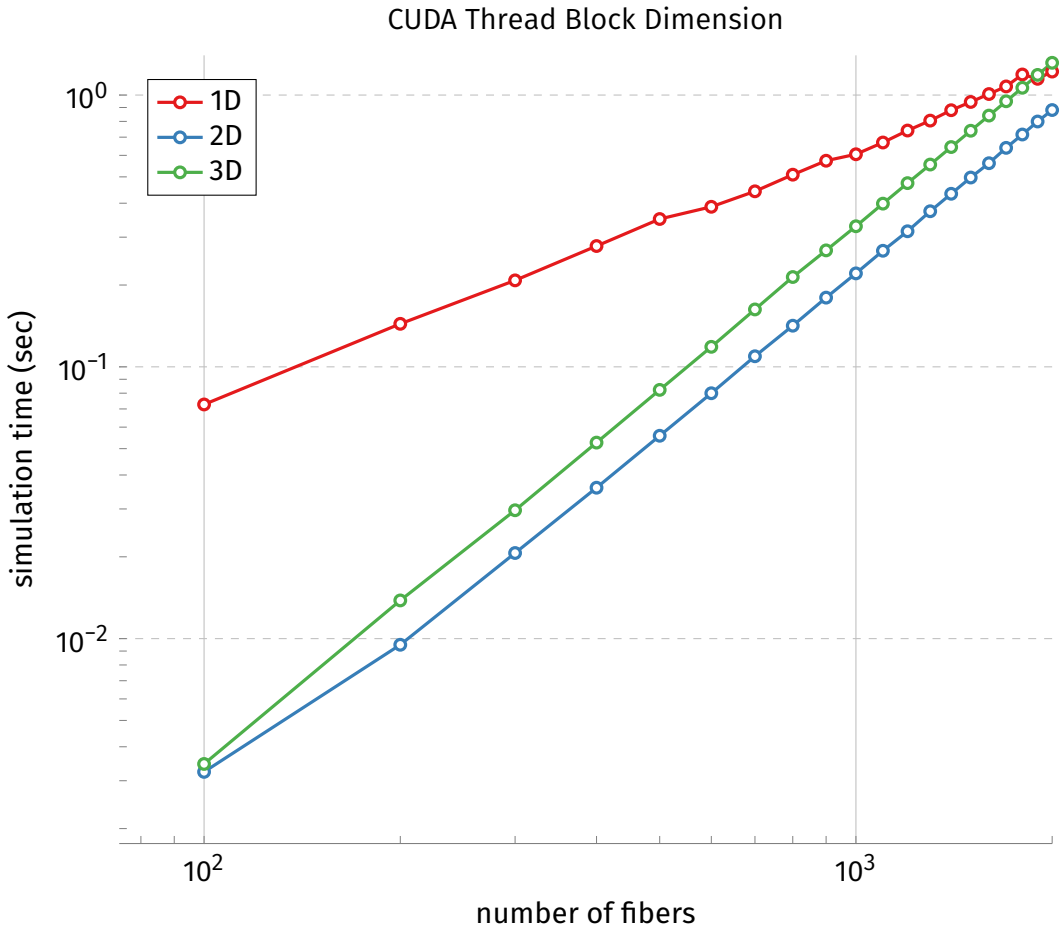


Figure 6.3: Benchmark of assemble system step for different thread block dimensions.

a two-dimensional thread block. The three-dimensional thread block is always slower and the performance gap grows with the number of fibers. The reason for this performance gap is the increased usage of atomics in the three-dimensional case. The overall usage of atomic functions can be inspected with the NVIDIA profiler *nvprof* and the profiling metric *Atomic Transactions*. This metric simply counts the total number of atomic transactions performed by calling atomic functions. Table 6.3 lists the *Atomic Transactions* counts for both the two-dimensional and three-dimensional simulation for a simple sedimenting sphere setup. The required *Atomic Transactions* in the three-dimensional case are almost two times larger than for the two-dimensional case. These additional transactions incur a performance penalty, because they serialize the access to memory and threads

have to wait while other threads finish writing to memory.

Algorithm	atomic_transactions
2D	1,269,325
3D	2,350,670

Table 6.3: Atomic transactions of 2D vs. 3D thread block dimensions.

Figure 6.3 also shows that the one-dimensional approach is slower than either the two-dimensional or three-dimensional approach. However, it appears to scale linearly whereas the other two scale exponentially. It can already be observed that the performance of the one-dimensional thread block becomes faster than the three-dimensional thread block for close to 2000 fibers. Unfortunately, the hardware of the workstation does not have enough memory to simulate more fibers, allowing the one-dimensional approach to overtake the two-dimensional approach. Thus at least for our simulation we always use the two-dimensional approach.

6.3 Linear Solvers

Next we compare the performance for different linear solvers. We explored two different types of solvers. The first type is a direct solver of the linear equations which is both implemented for OpenMP and the CUDA implementation. The second type are iterative solvers. Unfortunately, we can't make use of the iterative solvers main advantage to efficiently solve sparse matrices as the system for the rigid fiber simulation is a dense matrix. The time required for solving the linear system can be a very large part of the overall runtime, depending choice of solver and the fiber configuration as discussed in section 3.5. It is therefore very important to find the optimal solver for this particular solver to arrive at the best performing algorithm overall.

On the CPU side we used the direct solver provided by the OpenBLAS library, which is fully parallelized. For GMRES we used the single precision Fortran implementation from Frayssé et al. [4] which takes extensive advantage of the underlying BLAS functions parallelized by OpenBLAS. The original paper quotes that they choose GMRES because it was faster than a direct solver for their tests [14]. The benchmark results for this thesis are illustrated in Figure 6.4.

We observe exactly the same, GMRES is faster by a wide margin as reported

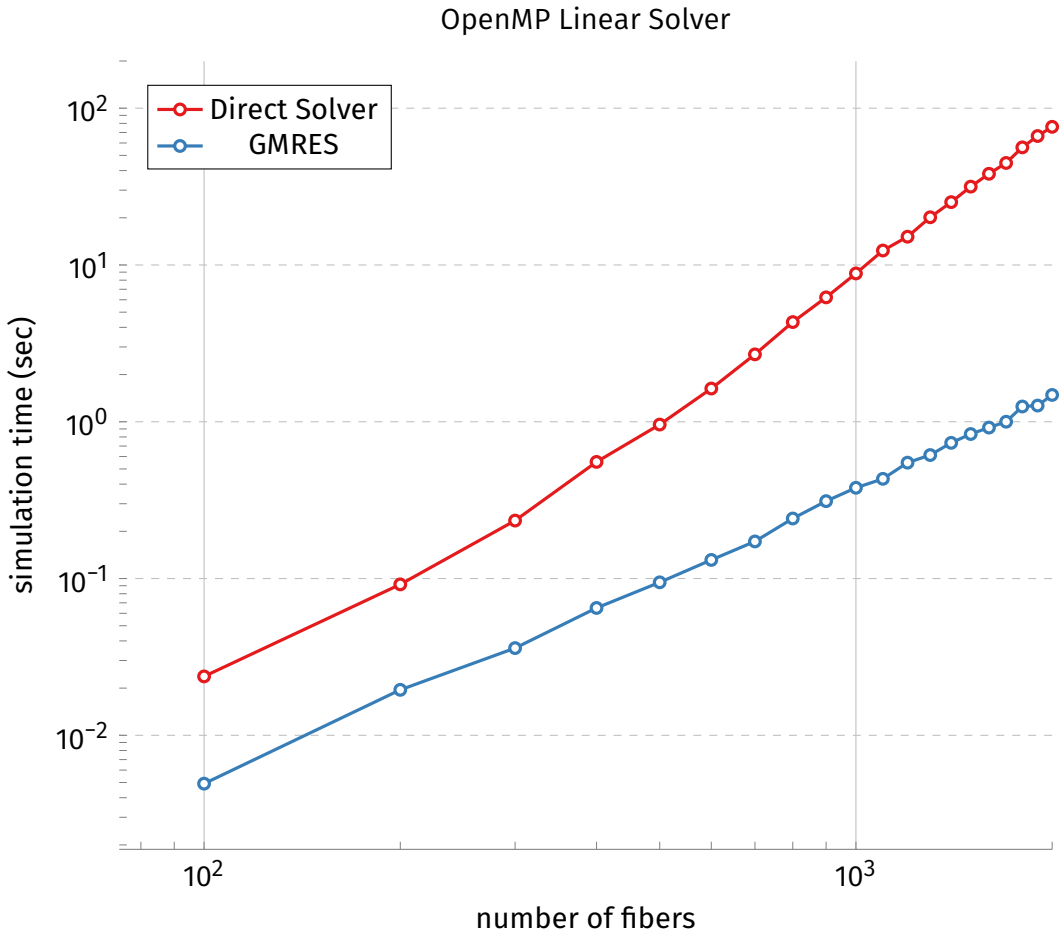


Figure 6.4: Benchmark of solve system step for different Fortran solvers.

by the original authors[14]. From an advantage of just 40× for 1000 fibers this increases to 300× for the maximum number of 2000. GMRES clearly scales better with the number of fibers than the direct solver. The observed number of iterations remain almost the same regardless of the number of fibers. However, there is one caveat to this clear victory, which is the minimal and average distance between the fibers. As their concentration increases so do the required GMRES iterations. It is likely that for a very complex setup with many more time steps, that at one point fibers come very close together. This might cause GMRES to have problems finding the solution. This effect will be tested later in section 7.2.

On the GPU side we used the direct solver provided by the MAGMA library. For the iterative solvers both GMRES as well as BiCGStab were compared as they can be easily exchanged and tested using the ViennaCL library. The benchmark results

for the CUDA solvers are illustrated in Figure 6.5.

Figure 6.5: Benchmark of solve system step for different GPU linear solvers.

The same fact that the iterative solvers are faster than the direct solver holds true for the GPU. However, compared to the CPU solvers the difference between the performance of the direct solver and the iterative solvers is not as large. At close to 2000 fibers the iterative solvers are only about 25× faster than the direct solver. Looking at the difference between the two iterative solvers, BiCGStab and GMRES they perform almost exactly the same. Any small differences can be attributed to small measuring uncertainties. Regardless of these clear results, it is always important to keep in mind, that this particular performance ratio only holds true for the specific fiber concentration which was benchmarked. For other concentrations the iterative solvers might need more iterations to find the

solution and might even perform worse than the linear solver.

6.4 Individual Steps

The next benchmark explores the relative time taken by each step of the algorithm, which were described in section 3.5 and 5.3. This gives valuable insight into the time allocation of the overall run time. Thereby helping to figure out which steps are the best ones to optimize. The results for the OpenMP implementation can be seen in Figure 6.6 and for CUDA in Figure 6.7. On the CPU the *Assemble System* and *Update Velocities* steps solve the inner integral in equation (3.3) analytically. On the GPU we use a purely numerical approach instead. This is done to use the fastest algorithm variation on both the CPU and GPU as we showed in section 6.2.1.

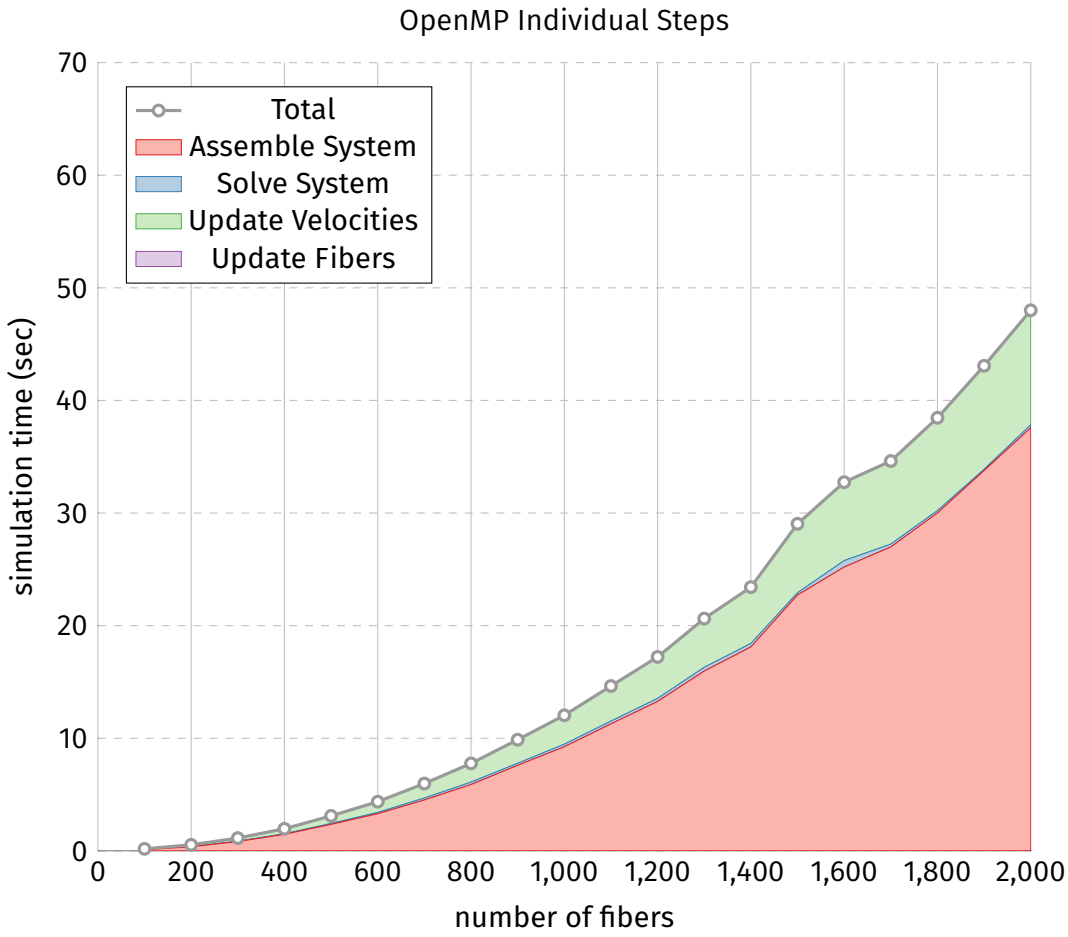


Figure 6.6: Benchmark for individual steps with OpenMP.

The results for the OpenMP implementation show that for 2000 fibers the *Assemble System* step is responsible for 78% and the *Update Velocities* step for 21% of the overall time. The two other steps *Solve System* and *Update Fibers* barely register with just 1%. However, as discussed in the previous section 6.3 this only holds true for sufficiently low concentrations of fibers.

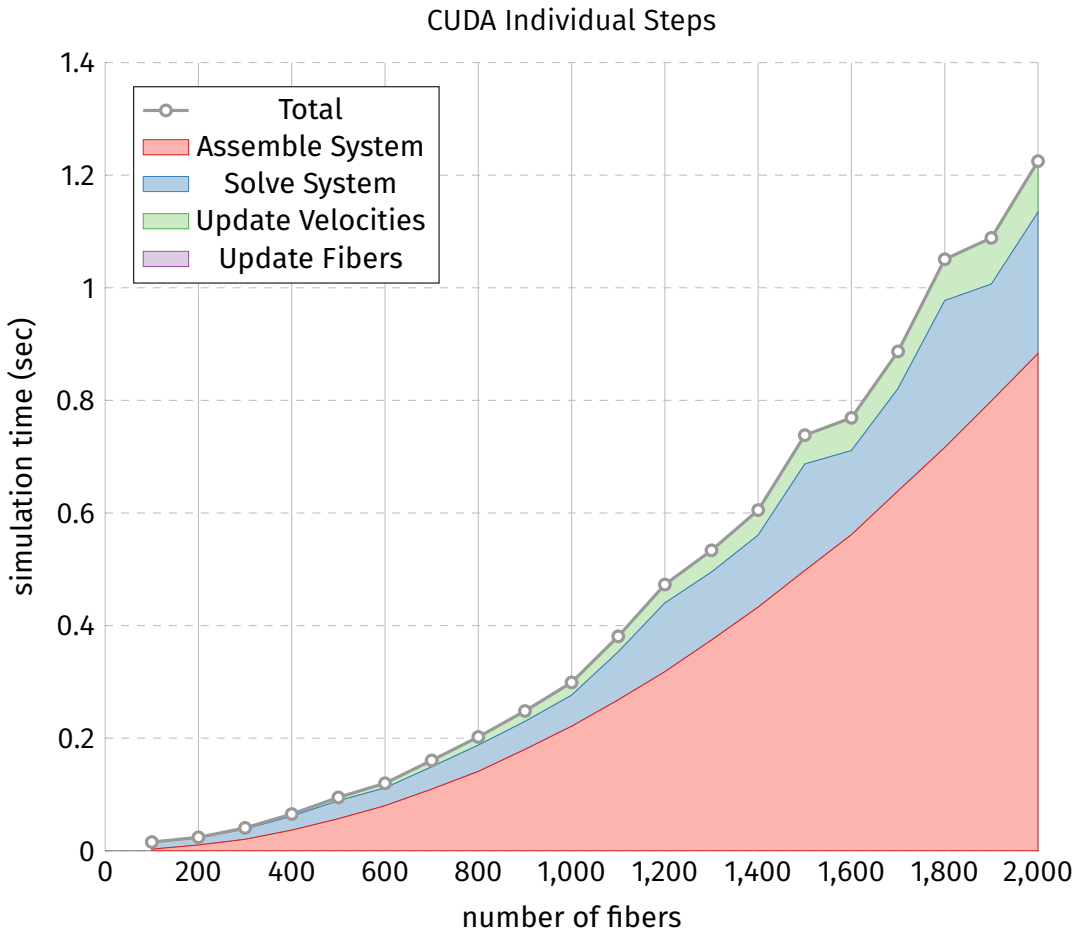


Figure 6.7: Benchmark for individual steps with CUDA.

For the CUDA implementation the *Assemble System* step remains the largest block with 72% of the time. However, now the *Solve System* step takes 20%. *Update Velocites* drops to just 7% and *Update Fibers* is negligible with below 1%.

This difference in the relative time between the steps can be attributed to the comparatively slow GMRES implementation on the GPU. In absolute terms for 2000 fibers both the CPU and GPU take 0.25 seconds for solving the system. But, when transferring the same relative distribution from the CPU to GPU this

makes the GPU GMRES implementation 40× too slow. The other steps all perform relatively better.

The reason for this most likely lies in the non-optimized code for the GMRES solver in ViennaCL. The authors openly state that the major goal for their library is easy of use and not pure performance [13]. The parallel BLAS functions from OpenBLAS on the other hand have been highly optimized and tested for a long time.

This point is reinforced if we look at the same numbers for the linear solvers. Here the absolute time for *Solve System* step on the GPU is just 7 seconds compared to the 76 seconds on the CPU. Again transferring the relative distribution from the CPU to the GPU yields just a small factor of 1.4×. This illustrates that the highly optimized code of the MAGMA library performs roughly on the same level as the optimized code from OpenBLAS.

6.5 GPU vs. CPU

The final benchmark compares the CPU and GPU performance. How to make a fair comparison of the simulation performance between the CPU and GPU is a hotly debated topic in the research literature[9][5]. The underlying architectures of the two approaches are completely different and thus hard to compare. Some try to extrapolate relative performance from the underlying FLOPs by taking processor count, frequency and memory bandwidth into account[9]. However, due to intricate hardware details this approach is also not applicable to all scenarios. Thus in the super computing community metrics like performance-per-dollar or even performance-per-watt have become the main focus[8].

Exploring this question in more detail is out of the scope of this thesis. Nevertheless we try to make a best effort to do fair comparison of CPU and GPU performance. In order to come as close as possible given these complexities and constraints, we used both a current CPU and GPU which can be considered a balanced system at the time of writing. Additionally, we implemented the parallel OpenMP version. It is directly based on the parallel CUDA version with the sole purpose to have as few difference between the two implementations as possible. Furthermore, the final benchmark for each uses the fastest possible algorithm variant as determined by all previously performed benchmarks to compare the best variant on each architecture.

The results for average time required to take a single time step is illustrated

in Figure 6.8. For OpenMP the algorithm uses the analytical integration of the inner integral. For CUDA, we used the numerical integration and the thread block dimension was chosen to be 2-dimensional.

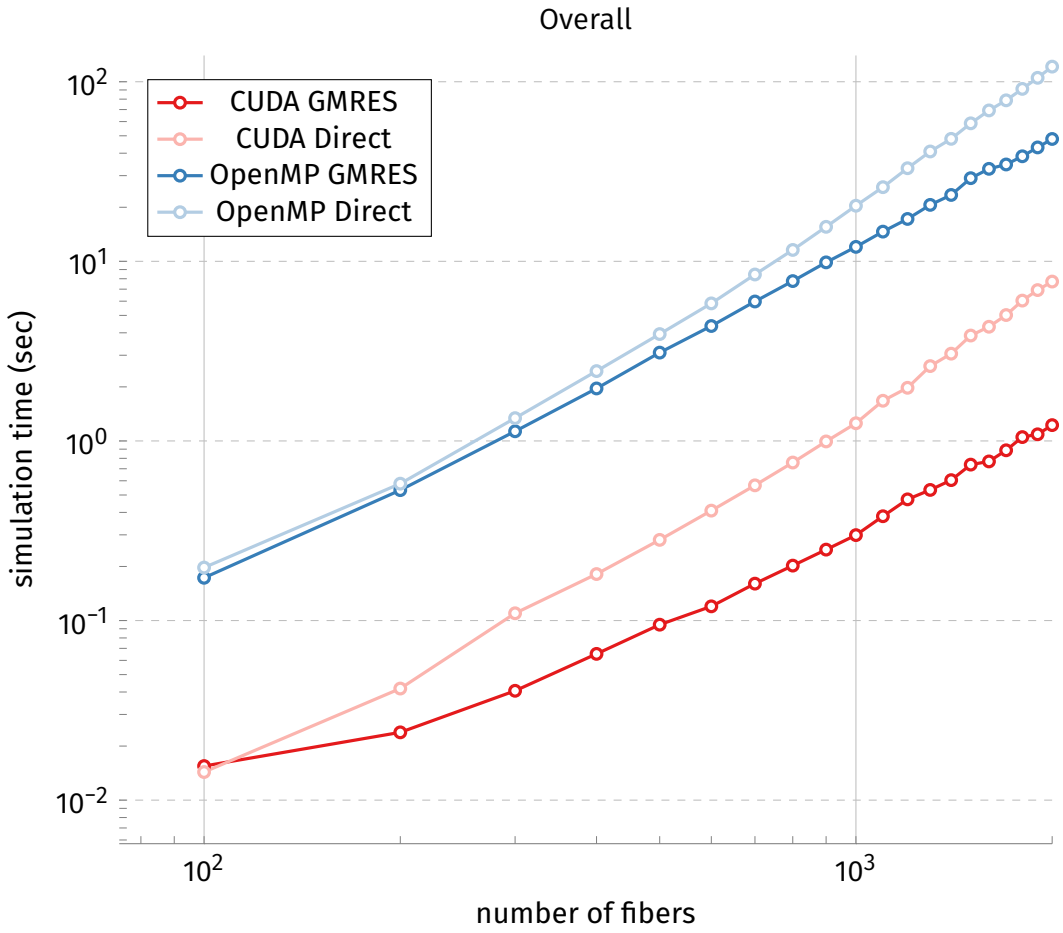


Figure 6.8: Benchmark of overall timestep for both OpenMP and CUDA.

The required simulation time on the GPU outperforms the CPU by a wide margin. The GPU version is faster for any number of fibers. The speedup factors for the presented simulation variations are listed in Table 6.4. CUDA maintains a relative performance of 40×. The only advantage the CPU version has is the potentially larger memory as 4GB on the GPU limits the number of fibers to roughly 2000, so for more fibers OpenMP is the only option.

We acknowledge that these numbers and performance increases are not necessarily fair. A different CPU and GPU combination from the one used in this thesis might perform differently. However, the relative performance should stay

M = 2000	OpenMP Direct	OpenMP GMRES	CUDA Direct	CUDA GMRES
OpenMP Direct	1×	—	—	—
OpenMP GMRES	3×	1×	—	—
CUDA Direct	16×	6×	1×	—
CUDA GMRES	99×	39×	6×	1×

Table 6.4: Overall speedup factor for 2000 fibers.

roughly the same. In the end, the only thing that really matters for the researcher working with rigid fibers is the time it takes to simulate large systems on the available workstation. There is no need to wait for computing time at a large computing cluster. Instead simulation can be run simply on a desktop computer allowing to rapidly iterate on the tests. The observed performance increase of 40× is a difference between a whole day of waiting for the simulation results and a quick 30 minute result during the day. The saved time also translates directly into the ability to simulate many more fibers than ever before. Before the largest simulation possible in a reasonable time frame where around 500 to 800 fibers, in contrast one single time step for 2000 takes at most 8 seconds on the GPU. Therefore the implementation on the GPU and the optimizations of the simulation code is of great value to the research of rigid fiber simulations.

Chapter 7

Experiments

The previous chapter showed how much faster the CUDA implementation is compared to the same algorithm implemented on the CPU. This significant increase in performance allows for new experiments.

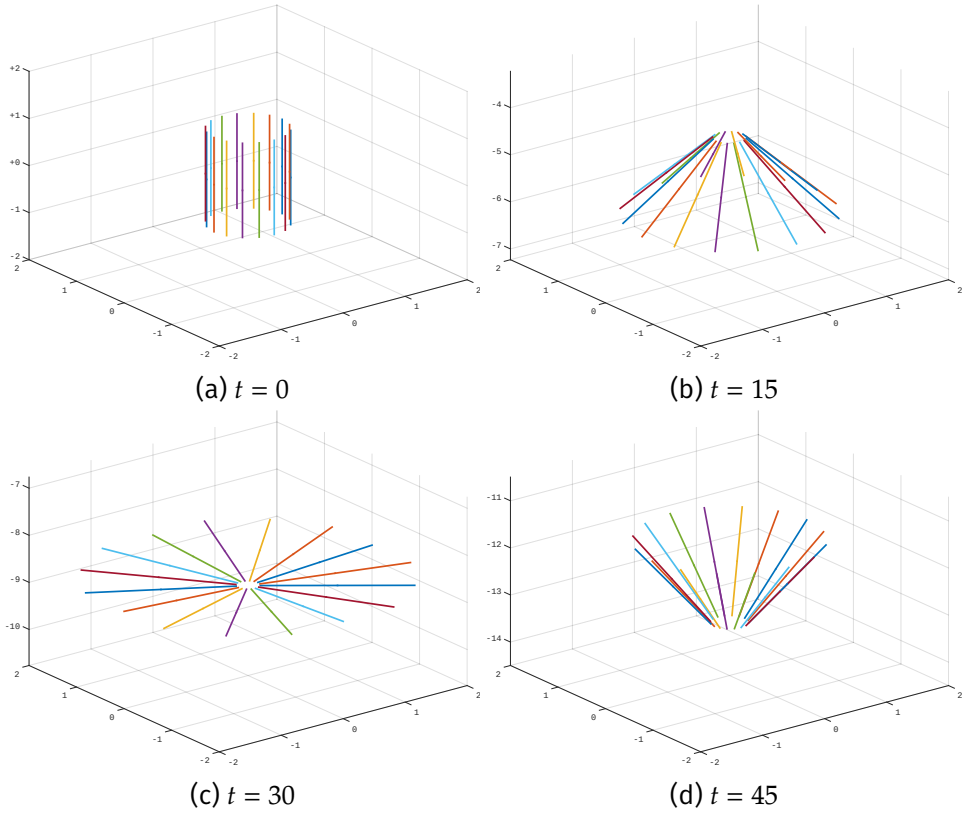
We will now perform a number of experiments to explore the numerical precision and validate the results against prior research. We begin by looking at tumbling orbits, a very delicate experiment requiring great precision. This is followed by a quick look at the effect of the fiber concentration on the required GMRES iterations to settle on the solution. Next we reproduce a very interesting physical phenomenon of letting a sphere of fibers sediment and validate our results against similar experiment by others. Furthermore, we will have a brief exploration of the effects of the number of fibers and the concentration of fibers on the sphere simulation. All experiments were run with the purely numerical, 2-dimensional CUDA algorithm implemented in the thesis. The systems were solved using the direct solver from MAGMA in order to avoid variations in the run time.

7.1 Tumbling orbits

The first example was used to verify that the single precision GPU version is able to replicate the result of the original double precision code. For this we looked at a very simple problem where a small number of fibers are set up with perfectly symmetrical positions and orientations. All fibers are evenly distributed on a circle and aligned vertically with gravity. During the simulation the fibers begin to rotate from their vertical orientation towards the horizontal position. Afterwards, they continue rotating back into the vertical position. This motion is referred to as

tumbling orbits and as long as there are no disturbances or numerical precision issues it repeat forever.

This simple but very interesting problem has also been simulated by Gustavsson and Tornberg [6]. Additionally an even more simplified version with only two fibers was studied both numerically and experimentally by Jung et al. [7]. This example is thus ideally suited to test and verify the numerical precision of the GPU simulation. A visualization of the GPU simulation of 16 fibers evenly distributed around a circle with a radius of 0.55, is shown in Figure 7.1.



(a) $t = 0$ (b) $t = 15$

(c) $t = 30$ (d) $t = 45$

Figure 7.1: Visualization Tumbling Orbits.

In order to analyze the periodic movement of the fibers we now look at the sedimentation velocity caused by gravity. In the initial setup the fibers are aligned vertically and are dropping with the maximum velocity. As they rotate into the horizontal orientation the velocity is decreasing and reaches its minimum once the fibers are perpendicular to the direction of gravity. Afterwards on their way back to vertical orientation the velocity increases again.

Figure 7.2 shows a graph of the sedimentation velocity of a single fiber over

time for both the single precision GPU code and the original double precision Fortran code. As the same force acts on all fibers and they perform the same motion they all have the same sedimentation velocity. For this particular setup the maximum velocity is 3.8 and the minimum velocity is 2.2. The graphs clearly shows the periodical rotation the fiber perform. This result perfectly captures the expected result obtained from prior simulation and experiments.

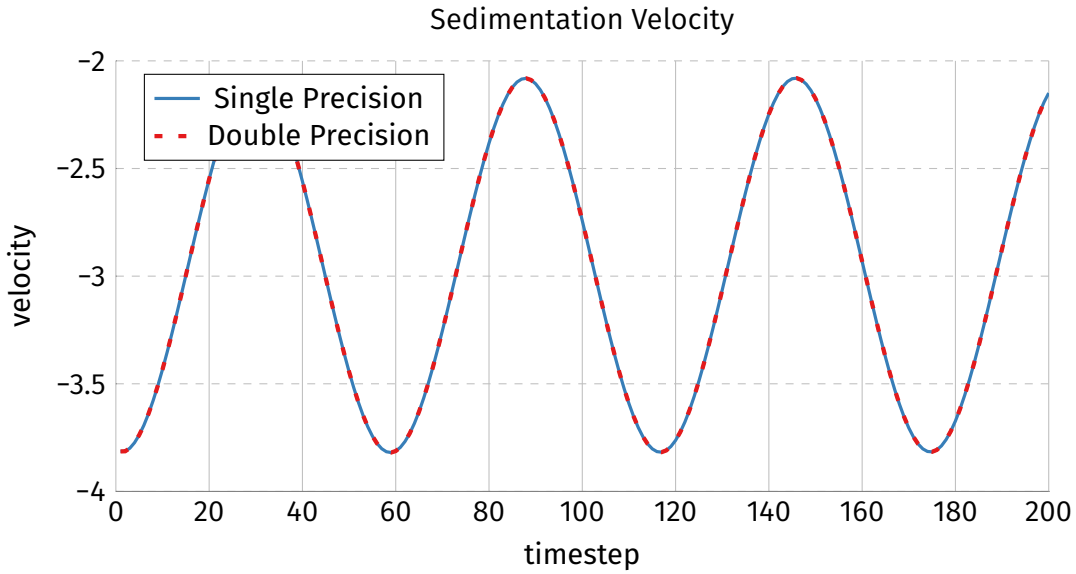


Figure 7.2: Comparison of sedimentation velocity for single- and double-precision simulation.

7.2 Fiber concentration effect on GMRES iterations

As alluded to in the discussion about the performance of direct solvers versus iterative solvers in section 6.3 we will now investigate how the concentration of fibers affects the GMRES iterations. In this context the concentration of fibers refers to the average distance between each fiber and its closest neighbor. Thus we use the average minimal pair-wise distance between the fibers as a measurement of fiber concentration. Figure 7.3 illustrates the number of iterations GMRS needs to solve the system for a given pair-wise distance. Please note that the x-axis as well as the y-axis are logarithmic to improve clarity.

We can see that for an increasing average pair-wise distance, i.e. a lower fiber concentration, the number of iterations decreases. The average iteration count

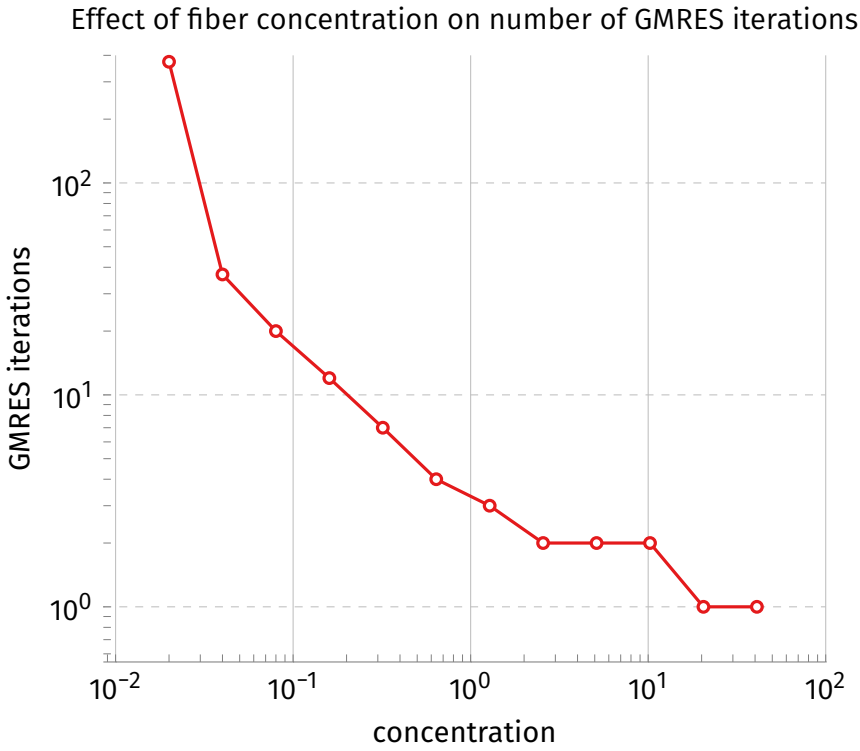


Figure 7.3: Effect of fiber concentration on GMRES iterations.

for the concentration (average distance of 0.4) we used during our benchmarks in chapter 6 lies between 3 and 10. However, as the average pair-wise distance decreases we observe an exponential increase in the number of iterations.

To find the reason for this we have again have to look at the integral in equation (3.3) which has to be solved for each fiber pair during the *Assemble System* step. The required Stokeslet computations in equation (2.6) include the distance between the fibers in the denominator. As the distance gets very small the value of the Stokeslet becomes large. These large terms can then cause the assembled matrix to become ill-conditioned and thus increase the number of iterations GMRES requires to settle on a solution.

This result is important to keep in mind when simulating long running complex fiber configuration. Here the probability that any two fibers get close to each other is very high. If this is the case solving the system using GMRES takes longer than expected. In practice it might thus be beneficial to switch to the direct solver, as it has a predictable runtime. This is especially true because the performance difference between direct and iterative solvers on the GPU is relatively small.

7.3 Sedimenting sphere

The next example showcases a more chaotic system with a large number of interacting fibers. This experiment is studied by several papers[10][12][1]. It is especially interesting because the observed results only occur if enough fibers are simulated. Our GPU simulation is able to efficiently handle up to 2000 fibers and is thus ideally suited for studying this example. For the experiment 2000 fibers are initially distributed inside a sphere with a concentration of 0.4. Both their positions and orientations are randomly generated inside this sphere. An illustration of an example run can be seen in Figure 7.4.

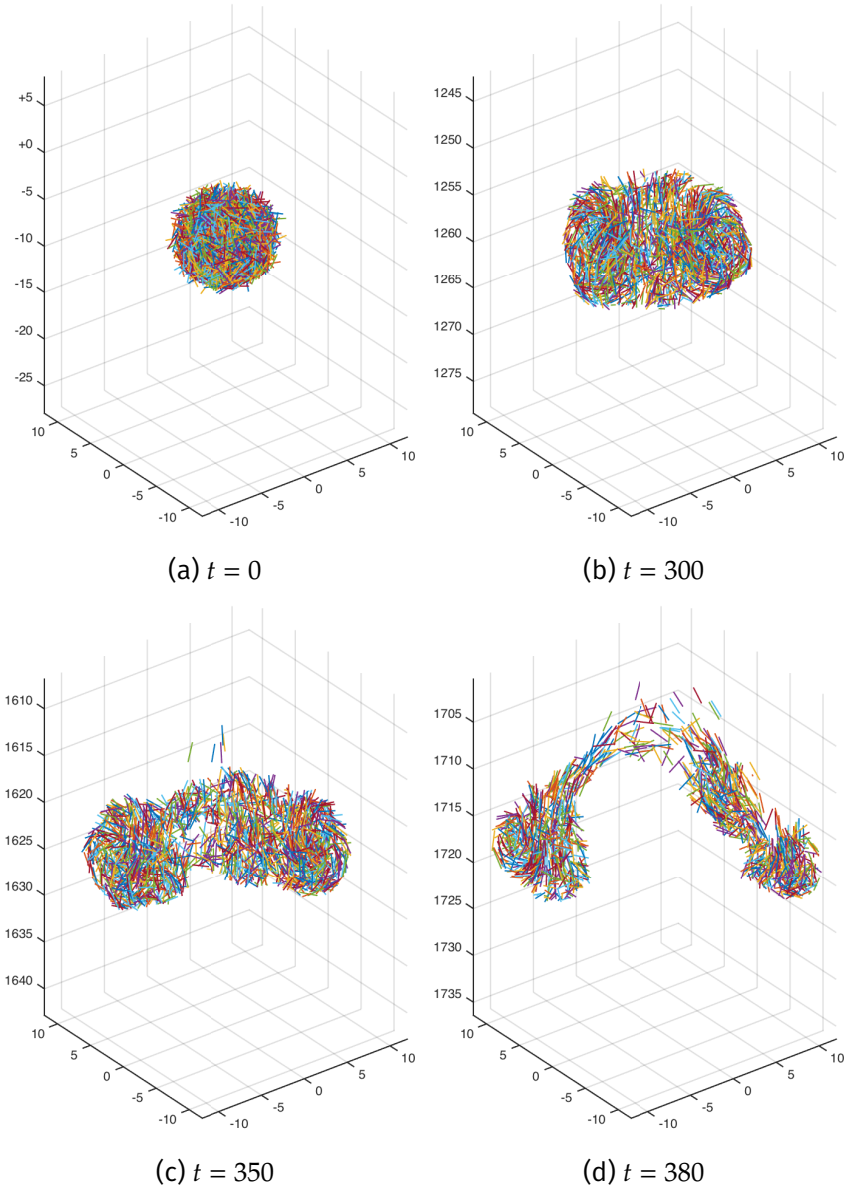
This sphere of packed fibers is then allowed to sediment due to gravity. Beginning from the spherical shape the interacting fibers slowly begin to form a continuously turning torus. Even though the behavior is more chaotic due to the large number of fibers and the random initial setup, this turning torus resembles the result for the tumbling orbits in section 7.1.

After some time the torus breaks and the fibers are split into multiple smaller clusters. Each cluster continues to sediment separately and slowly forms its own smaller torus again. However, due to tiny variations these toruses can be harder to see.

The calculation of the presented example only takes 7.7 seconds with the GPU simulation. Consequently it is possible to perform the 500 time steps of the simulation in a little bit over 1 hour. Simulating this many fibers in such a short time will allow new research of this interesting phenomenon. One interesting research question is for example determining what influences the stability of the torus.

To begin investigating this question we first define the stability of the torus in terms of the break up time. How to determine the exact break up time of the torus is quite challenging. This problem and question was also examined by Park et al. [12]. They define the break up as the moment when the torus starts to bend prior to actually breaking. Additionally, they describe an algorithm that determines which fibers are part of the torus. Consequently, they compute metrics such as the remaining fibers, the radius and velocity of the sedimenting torus. Unfortunately, they don't define a metric that determines the exact break up time. Thus we came up with our own measure of the break up time based on their work.

We use the same definition for the torus as described by Park et al. [12]. First we compute the initial radius R_0 of the sphere. Then as the sphere sediments and starts forming the torus a small number of fibers are separate. For each time step



(a) $t = 0$ (b) $t = 300$

(c) $t = 350$ (d) $t = 380$

Figure 7.4: Visualization Sedimenting Sphere.

the center of mass of all fibers belonging to the torus is calculated. Any fibers with a distance larger than R_0 in the direction of gravity are removed from the active set of fibers belonging to the torus. For the remaining fibers we then calculate the radius R of the torus in each direction and normalize it by the initial radius R_0.

We observe that the standard deviation of the radius in the direction of gravity continuously decreases as time passes. The standard deviation then reaches its minimum a few moments before the break up can be identified visually. We can now define this minimum of the standard deviation of the radius as a metric for the break up time. We verified this metric manually against many simulations and it showed great agreement with the visual break up. Using this metric we are now able to investigate the effect of some simulation parameters on the stability and break up time of the torus.

7.3.1 Fiber concentration effect on break up time

The first parameter we explored was the concentration of the fibers. Concentration again is defined in terms of the average distance between a fiber and its closest neighbor. For our experiments we fixed the number of fibers at 2000 and used the exact values for all other parameters only changing the concentration. Figure 7.5 indicates that there is a clear correlation between the fiber concentration and the time until breakup. This matches the results found by Park et al. [12].

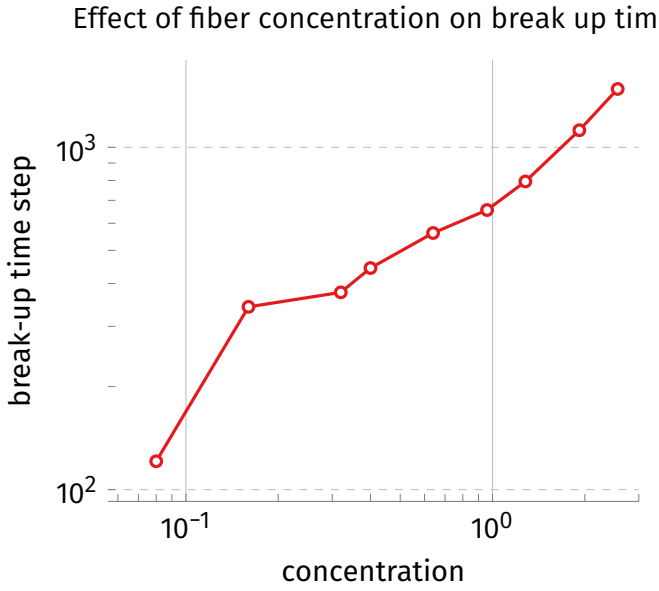


Figure 7.5: Effect of fiber concentration on torus break up time.

7.3.2 Number of fibers effect on break up time

The second parameter we looked at was the number of fibers. For this experiment only the number of fibers was changed. Every other parameter was fixed. For the concentration we chose the same value of 0.4 as during our benchmarks in chapter 6. The resulting chart is shown in Figure 7.6. It shows a positive correlation for the number of fibers and the break up.

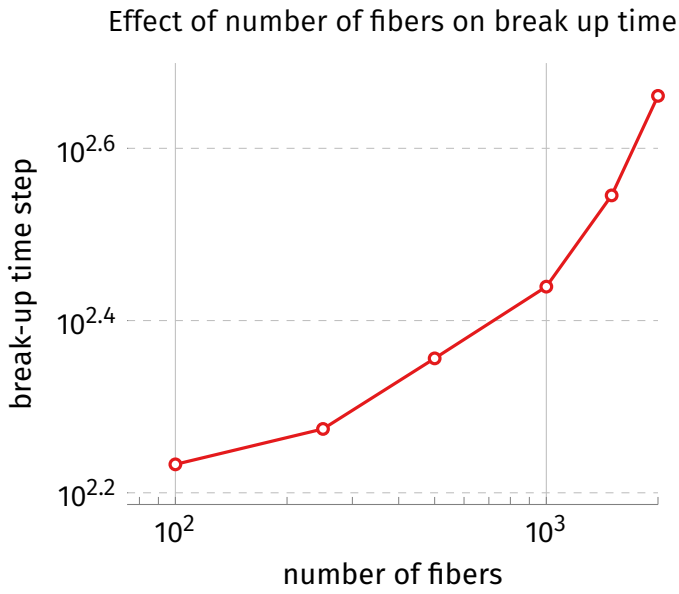


Figure 7.6: Effect of number of fibers on torus break up time.

These two results show that both the concentration and the number of fibers have an effect on the break up time of the torus. Future research should explore these correlations closer as our tests only looked at two parameters in isolation. How other parameters, like external forces, the numerical accuracy and the initial fiber distribution affect the stability of the torus are promising questions.

Chapter 8

Conclusion

In this thesis we have developed a completely new parallel GPU implementation for the numerical simulation of rigid fibers suspension using CUDA. This is faster and therefore many more fibers can be simulated than before. Furthermore being able to simulate more fibers enables the researcher to perform more extensive and in-depth studies of the various properties of the flow and simulation.

Based on the theoretical foundation of the simulated model and the original serial Fortran implementation we rewrote the algorithm to take advantage of massively parallel computational power available in modern GPUs. We outlined the required steps to implement the algorithm using CUDA. We investigated a number of different optimizations to further improve the performance. These explored optimizations included variations of the algorithm, GPU specific implementation details and the choice of linear solver. To reach the goal of comparing the performance between CPU and GPU the new parallel algorithm was backported to the CPU using OpenMP. Thereby improving the fairness of the comparison as much as possible.

During extensive benchmarking we showed that variations perform differently depending on the underlying architecture. Optimizations developed for the CPU implementation actually slow down the GPU implementation due to diverging execution branches. We also discovered that off-the-shelf GPU implementations of iterative solvers can be slower compared to their highly optimized CPU alternatives. Hopefully future research in this area will allow for even better performance of iterative solvers on the GPU.

Overall our GPU simulation on the NVIDIA GTX 970 is able to outperform the CPU version on an Intel Core i7 4770 by a factor of $20\times$ to $40\times$. This allows the GPU simulation to handle up to 2000 fibers, which is more than double then before,

and only takes 8 seconds to advance one time step on a desktop computer.

To test and validate our new parallel implementation, we performed a number of simulations of known experiments for sedimenting fibers. Using the test-case of tumbling orbits we showed that the numerical precision of our single precision implementation is able to reproduce the result obtained from the former double precision Fortran simulations. We also explored how the performance of iterative solvers is affected by the fiber concentration, showing that if fibers come too close the number of iterations increases and slows down the simulation. Our last experiment looked at the known phenomena that a sphere of fibers sedimenting form a torus and eventually break up into smaller clusters. Our results show an excellent agreement with both experimental and numerical work performed previously. Additionally, we showed that the stability of the torus is positively correlated with the increasing distance between fibers and the total number of fibers. Further research in this area is helped greatly by the fact that new case can rapidly be explored using our fast GPU simulation.

Currently the number of fibers in our simulation is limited by the amount of memory available on the GPU. In future we would like to research how to lift this restriction. One possible research direction is to avoid storing the matrix completely. Instead the required computation would have to be carried out iteratively and on-demand. This would allow for a practical unlimited number of fibers with the cost of an largely increased computation time. If this trade off is worth it remains to be seen. Figure Another exciting possibility is the move to a multi-device setup. Using multiple GPUs simultaneously would enable much larger systems, if the workload can be split efficiently. MAGMA, the library used for the GPU direct solver, offers a couple of interesting options in this area. The most challenging part to efficient parallelization is solving the system and broadcasting the result across multiple device. The other steps of the algorithm can be naturally partitioned and have already been implemented across multiple CPUs using OpenMPI.

Finally, to further improve the simulation it might be worth looking at the numerical algorithm itself. Adapting a different algorithm based on the fast summation method might not only improve the performance but also reduce the required memory allowing for faster and larger simulation. How this affects the accuracy and behavior of the simulation is an interesting future research question.

Bibliography

[1] F. Bülow, H. Nirschl, and W. Dörfler. “On the settling behaviour of polydisperse particle clouds in viscous fluids”. In: *European Journal of Mechanics - B/Fluids* 50 (Mar. 2015), pp. 19–26.

[2] Nvidia Corporation. *CUDA C Best Practices Guide*. 2014. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[3] Nvidia Corporation. *CUDA C Programming Guide*. 2014. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[4] V Frayssé et al. *A set of GMRES routines for real and complex arithmetics on high performance computers: Technical Report TR*. Tech. rep. PA/03/03, CERFACS, 2003.

[5] Chris Gregg and Kim Hazelwood. “Where is the data? Why you cannot debate CPU vs. GPU performance without the answer”. In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of systems and software*. IEEE, Apr. 2011, pp. 134–144.

[6] Katarina Gustavsson and Anna-Karin Tornberg. “Gravity induced sedimentation of slender fibers”. In: *Physics of Fluids* 21.12 (2009), p. 123301.

[7] Sunghwan Jung et al. “Periodic sedimentation in a Stokesian fluid”. In: *Physical Review E* 74.3 (2006), p. 035302.

[8] Shoaib Kamil, John Shalf, and Erich Strohmaier. “Power efficiency in high performance computing”. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE. 2008, pp. 1–8.

[9] Victor W Lee et al. “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 451–460.

[10] Bloen Metzger, Maxime Nicolas, and Élisabeth Guazzelli. “Falling clouds of particles in viscous fluids”. In: *Journal of Fluid Mechanics* 580 (May 2007), p. 283.

[11] J.D. Owens et al. “GPU Computing”. In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899.

[12] Joontaek Park et al. “A cloud of rigid fibres sedimenting in a viscous fluid”. In: *Journal of Fluid Mechanics* 648 (2010), pp. 351–362.

[13] K. Rupp, F. Rudolf, and J. Weinbub. “ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs”. In: *Intl. Workshop on GPUs and Scientific Applications*. 2010, pp. 51–56.

[14] Anna-Karin Tornberg and Katarina Gustavsson. “A numerical method for simulations of rigid fiber suspensions”. In: *Journal of Computational Physics* 215.1 (June 2006), pp. 172–196.