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

This Master's thesis project was carried out at the Numerical Analysis Group of the Department of Mathematics at the Royal Institute of Technology in Sweden.

First and foremost, I especially want to thank my supervisor Katarina Gustavsson for giving me the opportunity to come to Sweden and contribute to the field of particle suspensions. Her guidance and support was invaluable to the success of this thesis. Additionally, I am grateful to Michael Schliephake for his insightful comments and discussion on GPU programming. Finally, I want to thank Micheal Hanke for being my examiner and his continued support.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Predicting the physical behavior of particles suspended in fluids is of great interest in a variety of different fields. The ability to accurately model and simulate different categories of particle suspensions allows their properties to be analyzed and optimized for a large number of applications. Examples include medical applications were the delivery 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 characteristics of their material.

Even in the simplest flow cases, the flow of a particle suspension exhibits very complex and complicated dynamical behavior. The rheological properties of the suspension depend strongly on features such as the concentration of particles, particle shapes and particle interactions. In order to accurately capture the complex dynamics of a suspension using numerical simulations, a large number of particles are required in the simulation. Hence, the ability of the numerical algorithm to efficiently handle a large amount of particles is of crucial importance.

The work in this thesis focuses on increasing and optimizing the efficiency of a numerical algorithm for the simulation of sedimenting rigid and slender fibers in a viscous fluid. The fibers are modeled and simulated on a particle-level in a 3D free-space Stokes fluid. The mathematical model is based on a boundary integral formulation and a non-local slender body approximation by Tornberg and Gustavsson, [29].

There are many numerical studies of fiber suspension and several different methods have been developed for both rigid and flexible fibers. One approach is the so-called beads-model, where the fibers are modeled as a set of connected spherical beads (e.g. [3][10][27][31]). The immersed boundary method discretizes the fibers with Lagrangian markers and distributes the force onto a background grid which is then used to modify the fluid flow (e.g. [23][28]). Another approach is based on slender body theory which uses the large aspect ratio of the fibers to simplify the underlying model (e.g. [8][29][30]). A comprehensive review of the numerical studies of fiber suspension can be found in Guazzelli and Hinch, [7].

In this thesis we develop a high performance GPU implementation of the numerical algorithm for simulating fiber suspension. By taking advantage of the massively parallel architecture of modern GPUs many more fibers can be simulated compared to the previous CPU based implementation of the algorithm. The major goal is to easily and efficiently perform simulations on a high-end desktop computer or workstation readily accessible by the researcher. This will allow the researcher to explore the huge problem space and simulate a large number of fibers in a short amount of time. Thus, it enhances the capacity to rapidly iterate and discover interesting test cases. These cases can then be used as a starting point for large scale simulations using computing clusters.

The most costly part of the numerical algorithm is to account for the interactions between all fibers in the system. For the calculations we chose the ”naive” algorithm, where the interactions between every pair of fibers is computed. The development of such a naive algorithm is easier, more accurate and in some cases more cost effective than an alternative fast summation approach. A fast summation approach, like the fast multipole method, is both complex to implement and introduces a potentially large performance overhead. Therefore it might not result in the desired performance increase compared to the naive algorithm. Our choice of consumer-grade GPU hardware limits the simulation to single precision, thus maintaining the accuracy is very important. By using the naive algorithm on the GPU the goal is to strike a good balance.

This thesis is divided into the three major parts: *Previous Work*, *GPU Implementation* and *Results*. First we introduce the theoretical foundation of the numerical method in Chapter 2 and refer to the original paper by Tornberg and Gustavsson, [29], for in-depth details. The numerical algorithm and the serial implementation on the CPU developed in the original paper, [29], is presented in

Chapter 3. Together these two chapters form the *Previous Work* part of the thesis. Based on this the *GPU Implementation* part follows. Chapter 4 briefly introduces general purpose computing on the GPU. Combining the previous work and GPU computing, we then describe our new parallel implementation of the numerical algorithm on GPUs in Chapter 5. This represents the base for the efficiency and the performance improvements and is therefore the core of the thesis. Afterwards, the final *Results* are presented. The benchmarks in Chapter 6 illustrate the achieved performance increase of the parallel *GPU Implementation*. Finally, we perform and compare our simulation results to a few 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 a large number of fibers in order to generate the various patterns found in real world experiments.

In this chapter, we will present the theoretical foundation of the physics and the mathematical model 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 model is only presented in a reduced summary, and for more details we refer to Tornberg and Gustavsson, [29].

2.1 Stokes flow

We are interested in modeling the behavior of objects immersed in an incompressible fluid. We restrict ourselves to rigid bodies, which are slowly sedimenting in the fluid due to gravity. In the most general case, the flow can be modeled by the Navier-Stokes equations,

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

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 and ρ is the fluid density. Solving these equations is quite challenging due to their time dependence and non-linearity. However, in this work we are only concerned with slowly moving objects and the Reynolds number, $Re = \frac{\rho U L}{\mu}$,

is assumed to be very small. Given the constraint that $Re \ll 1$, the inertial and acceleration terms in the Navier-Stokes Eqns. (2.1) can be neglected and we arrive at the linear and time independent Stokes equations,

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

In addition to these equations we need two boundary conditions to solve the system. The first boundary condition accounts for the influence on the fluid from the presence of the objects. Here, 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 object surfaces and $\mathbf{u}_\Gamma$ the corresponding surface velocity. This condition constrains the fluid to have zero velocity relative to the object surfaces.

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 stating that the velocity field should be equal to a background velocity $\mathbf{U}_0$ at infinity,

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

In our simulations this background velocity is always set to 0. Through the motion of the immersed objects and the no-slip boundary conditions defined in Eqn. (2.3) the dependency on time is reintroduced in the Stokes Eqns. (2.2).

2.2 Boundary integral formulation

The Stokes equations are linear in both velocity and pressure, which allows them to be solved using a number of different methods for linear partial differential equations. The approach used in this thesis is the boundary integral method, see e.g. Pozrikidis [24].

2.2.1 Fundamental solutions

Starting from a boundary integral formulation of the Stokes equations, analytical solutions can be derived, so-called fundamental solutions. One such solution is

the Stokeslet. If $\mathbf{f}$ in Eqn. (2.2) is given by a point force acting at $\mathbf{y}$ with strength $\mathbf{F}$, i.e. $\mathbf{f} = \mathbf{F} \cdot \delta(\mathbf{x} - \mathbf{y})$, where $\delta(\mathbf{x} - \mathbf{y})$ is the Dirac delta function, the velocity field

$$u_i(\mathbf{x}) = \frac{1}{8\pi\mu} S_{ij}(\mathbf{x}, \mathbf{y}) F_j \quad i, j = 1, 2, 3, \tag{2.5}$$

where the tensor product

$$S_{ij}F_j = S_{i1}F_1 + S_{i2}F_2 + S_{i3}F_3, \tag{2.6}$$

is a solution to the Stokes equations. The term S_{ij} is the Stokeslet and is given by

$$S_{ij}(\mathbf{x} - \mathbf{y}) = \frac{\delta_{ij}}{|\mathbf{x} - \mathbf{y}|} + \frac{(x_i - y_i)(x_j - y_j)}{|\mathbf{x} - \mathbf{y}|^3}. \tag{2.7}$$

Later, we will need higher order fundamental solutions to the Stokes Eqns. (2.2). These can be obtained by simply differentiating the Stokeslet. One example is the so-called doublet

$$D_{ij}(\mathbf{x} - \mathbf{y}) = \frac{1}{2}\Delta S_{ij}(\mathbf{x} - \mathbf{y}) = \frac{1}{8\pi\mu}\left(\frac{\mathbf{I}}{|\mathbf{x} - \mathbf{y}|^3} - \frac{3((x_i - y_i)(x_j - y_j))^2}{|\mathbf{x} - \mathbf{y}|^5}\right). \tag{2.8}$$

2.2.2 Formulation for objects immersed in a fluid

Using the fundamental solution in Eqn. (2.5) we can now model the motion of immersed objects in a fluid. Assume that we have a total of M immersed rigid objects in the fluid. Each object m, for $m = 1, 2, \ldots, M$, is centered at $\mathbf{x}_c^m$ with an associated orthonormal basis $\mathbf{t}^m$ and surface Γ^m. Given a rigid body motion and the no-slip boundary condition in Eqn. (2.3) we can model the velocity field $\mathbf{u}(\mathbf{x})$ at each surface point $\mathbf{x} \in \Gamma^m$ of the object as

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

where $\mathbf{U}^m$ is the translational velocity and $\boldsymbol{\omega}^m$ is the rotational velocity of the object. This setup is illustrated for two objects in Fig. 2.1.

By combining rigid body motion, Eqn. (2.9), with a boundary integral formulation of the Stokes equations a relationship between the objects velocities and the force distribution, f, acting on the surface of the object is given by,

$$\mathbf{U}_i^m + (\boldsymbol{\omega}^m \times (\mathbf{x} - \mathbf{x}_c^m))_i = \frac{1}{8\pi\mu} \sum_{l=1}^{M} \int_{\Gamma^l} S_{ij}(\mathbf{x}, \mathbf{y}) f_j^l(\mathbf{y})\, dS_y \quad i, j = 1, 2, 3. \tag{2.10}$$

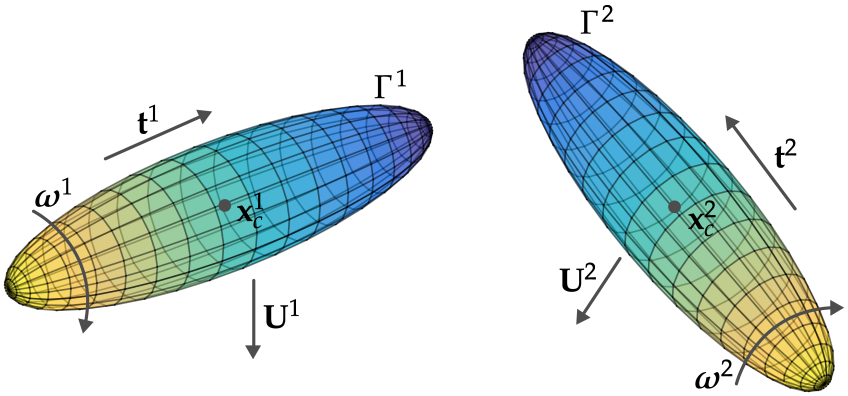


Figure 2.1: Two immersed objects in stokes flow, using a rigid body motion. Each fiber is characterized by the positions $\mathbf{x}^1, \mathbf{x}^2$ and the orientations $\mathbf{t}^1, \mathbf{t}^2$. The velocity of each point the surfaces Γ^1, Γ^2 is given by the rigid body motion described in Eqn (2.9).

For a sedimenting object, the translational and rotational velocities $\mathbf{U}^m$ and $\boldsymbol{\omega}^m$ as well as the force distribution $\mathbf{f}^m$ are unknown. In order to be able to solve the system we use the additional constraints

$$\mathbf{F}^m_{\text{object}} = \int_{\Gamma^m} \mathbf{f}^m(\mathbf{y})\, dS_y, \quad \mathbf{T}^m_{\text{object}} = \int_{\Gamma^m} (\mathbf{x} - \mathbf{x}_c^m) \times \mathbf{f}^m(\mathbf{y})\, dS_y, \tag{2.11}$$

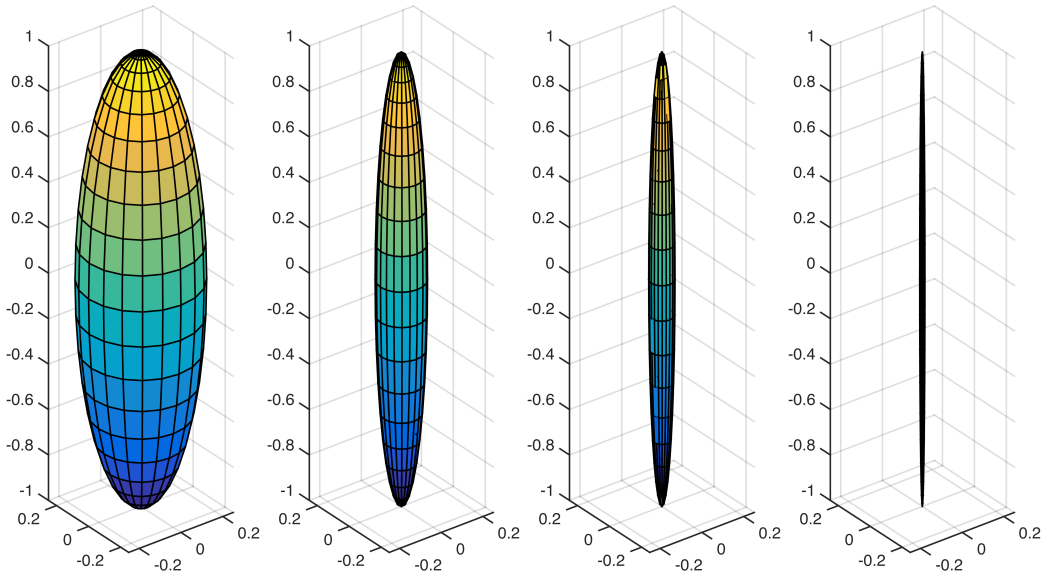
stating that the integrated force and torque over each object must be equal to the externally applied force and torque on the object. Given these additional constraints we are now able to solve Eqns. (2.10) and (2.11) for $\mathbf{f}^m$, $\mathbf{U}^m$ and $\boldsymbol{\omega}^m$ for all fibers $m = 1, 2, \ldots, M$. Once we have the velocities, $\mathbf{U}^m$ and $\boldsymbol{\omega}^m$, we can update the position and orthonormal basis of each object by solving,

$$\frac{d}{dt}\mathbf{x}_c^m = \mathbf{U}^m, \quad \frac{d}{dt}\mathbf{t}^m = \mathbf{t}^m \times \boldsymbol{\omega}^m. \tag{2.12}$$

The integral appearing in the right-hand side of Eqn. (2.10) must in most cases be evaluated by numerical quadrature.

2.3 Slender fibers

The formulation developed in the previous section will now be adapted for a rigid fiber suspension. Consider a straight, rigid body of length $2L$ and radius a, and let $\epsilon = a/2L$ denote a slenderness parameter. If $\epsilon \ll 1$ the body is referred to as a slender body (slender fiber) as illustrated in Fig. 2.2.



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

Figure 2.2: Illustration of the slenderness parameters. When ϵ becomes smaller the shape of the body asymptotically approaches that of an infinitesimal slender fiber.

Our goal is to simulate many slender fibers. However, using the 2D boundary integral formulation will be very expensive to solve numerically. As the aspect ratio $1/\epsilon$ of the fiber increases, the number of quadrature points must increase in order to accurately resolve the flow field around the fibers. However, for slender bodies a slender body approximation can be used instead.

The slender body approximation is derived from the boundary integral formulation of the Stokes Eqns. (2.10). Using asymptotic analysis the governing surface integrals are reduced to 1D integral equations along a centerline of the fiber. By matching the fluid velocity at a virtual boundary of a slender ellipsoid to the fiber centerline velocity, see Götz, [5]. The accuracy of this approximation is in $O(\epsilon)$. The reduction in dimensionality from 2D boundary integral equations to 1D integral equations is crucial for the ability to include a large number of fibers in the simulations.

The slender body approximation yields a coupled system of 1D integral equations over two fundamental solutions of the Stokes equations, the Stokeslet, Eqn. (2.7) and the Doublet, Eqn. (2.8). The coupled system relates the forces exerted on the fibers to their velocities and captures the non-local interaction of the fiber with itself (as mediated by the fluid), as well as with any other structures

with the fluid, such as other fibers or external boundaries. The system of integral equations is solved using a boundary integral method. Details of the model are given in Tornberg and Gustavsson, [29].

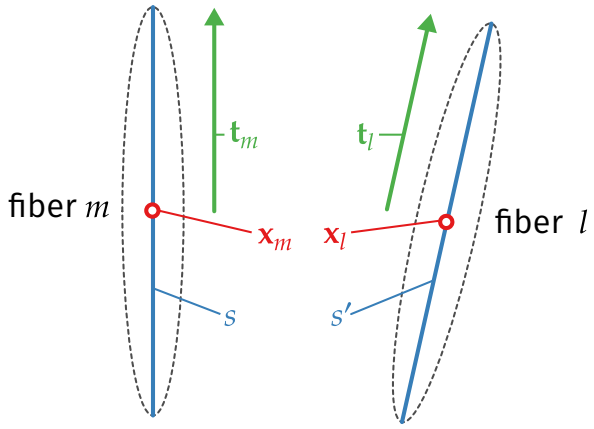


Figure 2.3: Slender fiber approximation. Each fiber is described by the center points $\mathbf{x}_m, \mathbf{x}_l$ the orientations given by the unit tangent vectors $\mathbf{t}_m, \mathbf{t}_l$ and the center line parameterized by s and s'. The points on each center line is thus given by $\mathbf{x}_m(s) = \mathbf{x}_m + s\mathbf{t}_m$.

Assume that we have M fibers immersed in the fluid. As shown in Fig. 2.3 each fiber is now defined by its centerline and parameterized by the arc-length, $s \in [-L, L]$. For fiber m the coordinates of the centerline are given by $\mathbf{x}_m(s,t) = \mathbf{x}_m(t) + s\mathbf{t}_m(t)$, where $\mathbf{x}_m$ is the center point and $\mathbf{t}_m$ the unit tangent vector of the fiber and $m = 1, 2, \ldots, M$.

If the fluid exerts a force per unit length, $\mathbf{f}_m$ on fiber m, the slender body approximation for the velocity of the centerline of fiber m can be stated in non-dimensional form as

$$\begin{aligned} d(\dot{\mathbf{x}}_m + s\dot{\mathbf{t}}_m) = \left[d(\mathbf{I} + \mathbf{t}_m\mathbf{t}_m^\top) + 2(\mathbf{I} - \mathbf{t}_m\mathbf{t}_m^\top)\right]\mathbf{f}_m(s) \\ + (\mathbf{I} + \mathbf{t}_m\mathbf{t}_m^\top)\bar{\mathbf{K}}[\mathbf{f}_m](s) + \mathbf{V}_m(s). \end{aligned} \tag{2.13}$$

Here d is a geometry parameter,

$$d = -\ln \epsilon^2 e, \tag{2.14}$$

and $\bar{\mathbf{K}}[\mathbf{f}_m](s)$ is an integral operator given by

$$\bar{\mathbf{K}}[\mathbf{f}_m](s) = \int_{-1}^{1} \frac{\mathbf{f}(s') - \mathbf{f}(s)}{|s' - s|}\, ds'. \tag{2.15}$$

The contribution to the velocity of fiber m from the other fibers in the system is accounted for in $\mathbf{V}_m(s)$ as

$$\mathbf{V}_m(s) = \sum_{\substack{l=1 \\ l\neq m}}^{M} \int_{-1}^{1} \mathbf{G}(\mathbf{R}_{lm}(s,s'))\mathbf{f}_l(s')\,ds', \tag{2.16}$$

where $\mathbf{R}_{lm}(s,s') = \mathbf{x}_m + s\mathbf{t}_m - (\mathbf{x}_l + s'\mathbf{t}_l)$ is the distance between one point on fiber m and one point on fiber l as illustrated in Fig. 2.4.

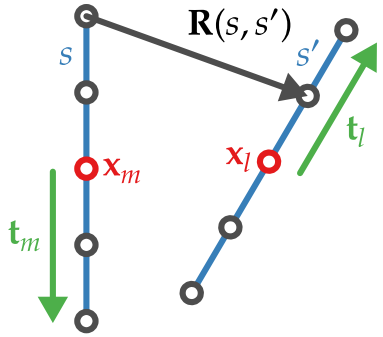


Figure 2.4: Two fibers with five quadrature points each. The interaction between two fibers is given by the term $\mathbf{V}_m$ and depends solely on the distance between the two. The integral in Eqn. (2.16) can not be evaluated analytically and will be approximated by a quadrature rule.

The Green's function in this case is a linear combination of two fundamental solutions and reads

$$\mathbf{G}(\mathbf{R}) = \begin{cases} \mathbf{S}(\mathbf{R}) + 2\epsilon^2\mathbf{D}(\mathbf{R}) & \text{if } l \neq m \\ 0 & \text{if } l = m \end{cases} \tag{2.17}$$

with $\mathbf{S}$ and $\mathbf{D}$ as defined in Eqns. (2.7) and (2.8).

In the non-dimensionalization the half-length of the fiber has been used as a characteristic length, $L_c = L$. This will give us a characteristic velocity and time as

$$U_C = \frac{d\Delta\rho g V}{4\pi\mu_f L}, \quad T_C = \frac{2\pi\mu_f L^2}{d\Delta\rho g V}. \tag{2.18}$$

The unknowns in Eqns. (2.13) are the translational and rotational velocities, $\dot{\mathbf{x}}_m$ and $\dot{\mathbf{t}}_m$ and the force distribution along the fiber $\mathbf{f}_m(s)$. To close the formulation we use the additional conditions stating that the integrated force and torque on each fiber must balance the external forces and torques applied to the fibers,

$$\mathbf{F}_m = \int_{-1}^{1} \mathbf{f}_m(s)\,ds = \mathbf{F}_g, \quad \mathbf{T}_m = \int_{-1}^{1} s(\mathbf{t}_m \times \mathbf{f}_m(s))\,ds = 0. \tag{2.19}$$

For our simulation the external force is just gravity and the external torque is set to zero.

Chapter 3

Numerical algorithm and serial implementation

In the previous 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 of boundary integral formulations and the slender body approximation to efficiently model the behavior of rigid fibers. Using this background we will now review the numerical approach used for the simulation.

We will separate the overall algorithm into four steps and discuss each step 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 Discretization

In Sec. 2.3 we derived a closed system of equations describing the motion of slender fibers in a fluid, see Eqns (2.13) and (2.19). In order to solve these equations we have to discretize them. For this we start by expanding the force as a sum of $N+1$ Legendre polynomials, $P_n(s)$, as

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

where the coefficients $\mathbf{a}_m^n$ are unknown vectors with three components, for each direction in space. The number of Legendre polynomials, N, used in the force expansion is a numerical parameter and is set to 5 in our simulation.

By algebraic manipulation of the Eqns. (2.13) and (2.19) and the use of the orthogonality properties of the Legendre polynomials, a linear system of equations for $\mathbf{a}_m^n$ for $m = 1, 2, \dots, M$ and $n = 1, 2, \dots, N$ which only includes computational quantities, can be obtained. Furthermore, two separate equations for the translational and rotational velocities of the fibers are obtained. They are given by

$$\dot{\mathbf{x}}_m = \frac{1}{2d}\left[d(\mathbf{I} + \mathbf{t}_m\mathbf{t}_m) + 2(\mathbf{I} - \mathbf{t}_m\mathbf{t}_m)\right]\mathbf{F}_m + \frac{1}{2d}\int_{-1}^{1}\mathbf{V}_m(s)\,ds, \tag{3.2}$$

$$\dot{\mathbf{t}}_m = \frac{3}{2d}(\mathbf{I} - \mathbf{t}_m\mathbf{t}_m)\int_{-1}^{1} s\mathbf{V}_m(s)\,ds. \tag{3.3}$$

Once the linear system of equations has been solved for $\mathbf{a}_m^n$, the forces on the fibers can be computed using Eqn. (3.1). Using the forces, the position and orientation of the fibers can be updated by integrating equations (3.2) and (3.3) in time. For more details and in-depth discussions please refer to the original paper by Tornberg and Gustavsson, [29]. Below we will describe the numerical algorithm developed to solve the problem.

3.2 Assemble System

The first step of the algorithm is to compute and assemble the linear system of equations. In the *Assemble System* step all interactions between the fibers are computed. This is a very time consuming part of the algorithm since the computational cost is of $O(M^2)$.

Writing the system in a standard form $\mathbf{A}\bar{\mathbf{a}} = \mathbf{b}$ gives the following structure of the dense $3MN \times 3MN$-matrix $\mathbf{A}$ and the right-hand side $\mathbf{b}$,

$$\mathbf{A} = \begin{bmatrix} \mathbf{I} & \bar{A}_{12} & \cdots & \bar{A}_{1M} \\ \bar{A}_{21} & \mathbf{I} & \cdots & \bar{A}_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{M1} & \bar{A}_{M2} & \cdots & \mathbf{I} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \bar{b}_1 \\ \bar{b}_2 \\ \vdots \\ \bar{b}_M \end{bmatrix}. \tag{3.4}$$

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

Inner integral For each $\bar{A}_{ml}$, a 3×3 matrix, Θ_{lm}^{kn}, where

$$\Theta_{lm}^{kn} = \int_{-1}^{1} \left[\int_{-1}^{1} \mathbf{G}(\mathbf{R}(s, s'))P_k(s')\, ds' \right] P_n(s)\, ds, \tag{3.5}$$

has to be evaluated for each force index $k, n = 1, 2, \ldots, N$. $\mathbf{G}$ and $\mathbf{R}$ are defined as in Eqns. (2.17) and (2.16), respectively. A similar term has to be evaluated for the right-hand side $\mathbf{b}$. One approach for evaluating the integrals is to use a standard Gaussian quadrature for both the inner and outer integral.

Another option is to use 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 Tornberg and Gustavsson, [29]. 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 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 does not have a negative effect on the performance.

Choosing between both options requires a careful examination of the accuracy and performance trade-off. 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.

Whenever we evaluate the integrals using numerical quadrature, 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.

3.3 Solve system

After having assembled the linear system $\mathbf{A}\bar{\mathbf{a}} = \mathbf{b}$ the next step to solve it. This can be done using standard linear equation solvers.

The linear system can either be solved using a direct solver or an iterative method like GMRES. As long as the fibers are not too close to each other the matrix is well-conditioned and 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 Sec. 6.3.

3.4 Update velocities

The force coefficients obtained by solving the linear system can now be used to calculate the right hand side in the equations for $\dot{\mathbf{x}}_m$ and $\dot{\mathbf{t}}_\mathbf{m}$, Eqns. (3.2) and (3.3). Here, the required implementation is similar to the implementation of the *Assemble System* step.

3.5 Update fibers

The final step takes care of advancing the fibers forward in time by solving Eqns. (3.2) and (3.3). These equations do not impose any strict stability restrictions, so an explicit time-stepping scheme can be used. We use the same second order multi-step method as 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

$$\frac{3\mathbf{x}_m^{i+1} - 4\mathbf{x}_m^i + \mathbf{x}_m^{i-1}}{2\Delta t} = (2\dot{\mathbf{x}}_m^i - \dot{\mathbf{x}}_m^{i-1}), \tag{3.6}$$

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 first order forward Euler method. Solving for the orientation vector, $\mathbf{t}_m$, is done using the same discretization. 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 the *Update Fibers* 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.

3.6 Algorithm summary

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

At each time step t^n, given the fiber positions, $\mathbf{x}_m^n$, and orientation $\mathbf{t}_m^n$

1. **Assemble system**

   This step assembles the matrix, $\mathbf{A}$, and the right hand side, $\mathbf{b}$ of the linear system of equations. For the assembly we have to evaluate Θ_{lm}^{kn} in Eqn. (3.5) for all fibers in the system.

2. **Solve system**

   This step solves the linear system of equations for the coefficients in the force expansion, Eqn. (3.1).

3. **Update velocities**

   Using the forces from previous step, this step computes the velocities of the fibers by computing the right hand side of Eqns. (3.2) and (3.3).

4. **Update fiber postions and orientations**

   In this step the fiber positions and orientations are updated by integrating Eqns. (3.2) and (3.3) in time. This step yields the fiber configuration at time step t^{n+1}, i.e. $\mathbf{x}_m^{n+1}$, and $\mathbf{t}_m^{n+1}$.

For the number of fibers used in this work (500–2000), empirical results show that the majority of the required computation time is spent on the *Assemble System* step and on the *Update Velocities* step. The time required for advancing the simulation state in the *Update Fibers* step is completely negligible. In Chapter 6, we will see that the same holds true for the new parallel implementation.

3.7 Remarks concerning GPU implementation

There are some remarks to be made before we turn our attention to the GPU implementation of the algorithm. The most important step to optimize is the *Assemble System* step, since it is the most time consuming step. 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. As the required computations for the *Update Velocities* step are similar to the computations made in the *Assemble System* step, it will also benefit from the optimizations. Additionally, we will look at how the two different options for solving the integral in Eqn. (3.5), either combined analytical and numerically or purely numerical, perform in the parallel environment.

Since we will not be writing our own implementation of linear solvers on the GPU we have only limited influence on the performance of the *Solve System* step. As we instead treat it as a black box we have to rely on the efficiency of pre-existing libraries. The only thing we can control is the choice of which library and solver to use. For direct solvers the computational time only depends on the number of unknowns and is approximately constant throughout a 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.

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 used for running arbitrary code instead of being limited to graphics related computations. This allows for 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., [21].

The deciding factor for the performance is how well the application can be parallelized in order to take advantage of the massively parallel architecture of GPUs. This massive parallelism has lead to potentially large performance advantages of GPUs over CPUs as illustrated in Fig. 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 units, 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.

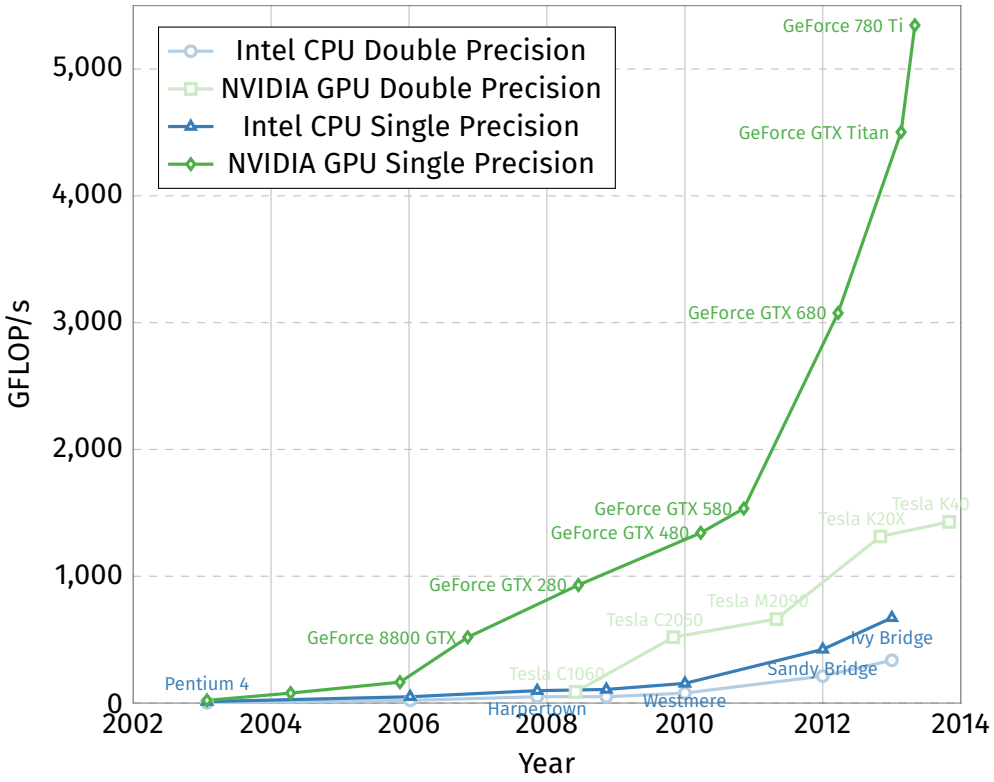


Figure 4.1: Theoretical GFLOP/s. Year over year increase in theoretical floating-point operations per second for CPUs and GPUs, [20].

The huge difference in performance between a GPU and a CPU mainly derives from the number of independent compute units. How a compute unit is exactly defined differs between architectures and vendors. On a CPU the number of compute units usually refers to the number of CPU cores. Each individual CPU cores is very fast, however, CPUs usually only have four, eight or maybe sixteen cores. In contrast to this, GPUs can have several hundreds of independent compute units. Each GPU compute unit can perform calculations in parallel and thus provides the opportunity to yield big performance improvements for high-throughput type computations. This is one reason why general purpose computing on GPUs was introduced to the world of supercomputers. Over time, a growing number of

supercomputers started supplementing their computing 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, [13]. 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.

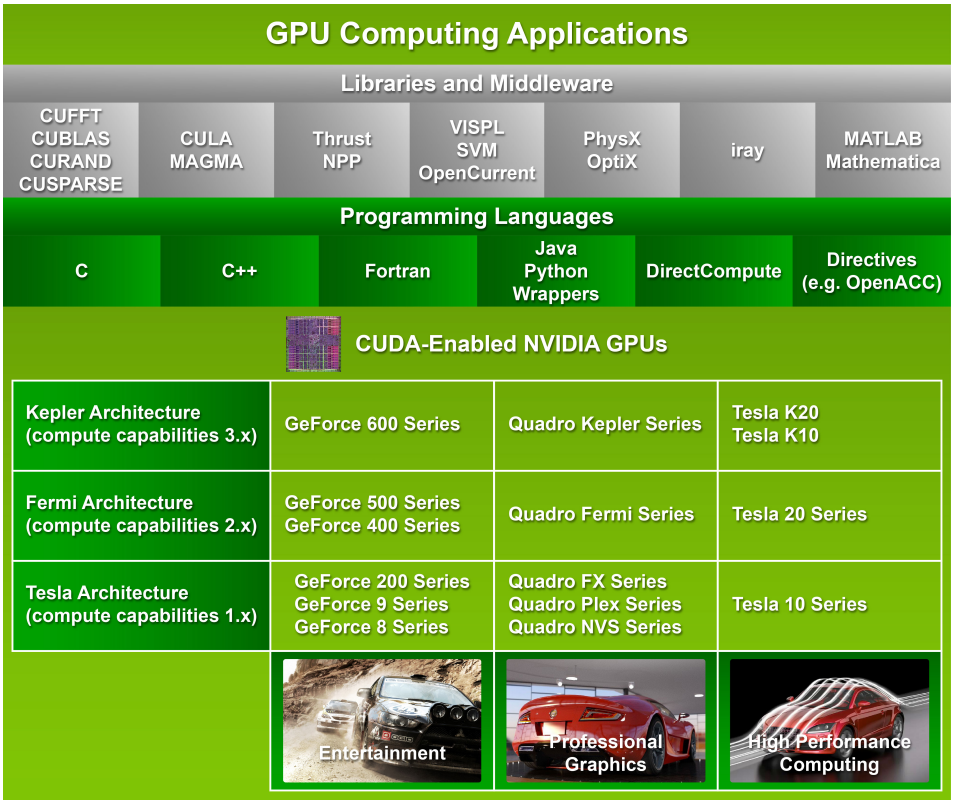


Figure 4.2: Overview of the CUDA platform, [20], describing the available libraries, programming environments and application of general purpose computing using CUDA.

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 use 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. Fig. 4.2 illustrates the different components of the overall CUDA platform.

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 units of the GPU. To illustrate how a very basic kernel invocation looks, Listing 4.1 shows a code sample for a very simple vector addition.

```
1  // Kernel definition
2  __global__ void VecAdd(float *A, float *B, float *C)
3  {
4    int i = threadIdx.x;
5    C[i] = A[i] + B[i];
6  }
7
8  int main()
9  {
10   ...
11   // Kernel invocation with N threads
12   VecAdd<<<1,N>>>(A,B,C);
13   ...
14 }
```

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 avoid any order-dependent operations 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 units of the GPU, CUDA defines a thread hierarchy. As discussed previously a GPU consists of many independent compute units. On NVIDIA GPUs these units 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 distributed to all the available SMs, which allows for automatic scalability depending on the

number of SMs available as illustrated in Fig. 4.3. Thus the developer only has to divide the workload into appropriately sized blocks of threads and invoke the kernel.

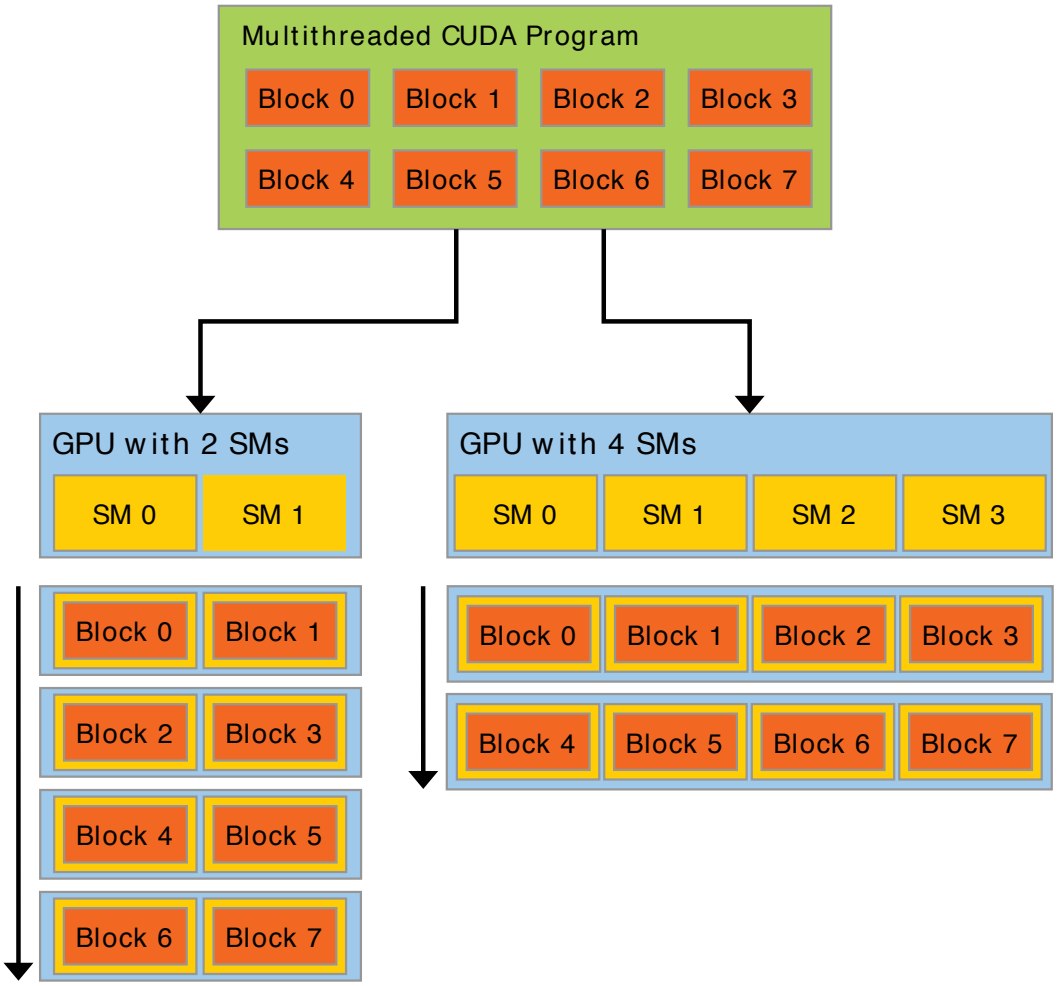


Figure 4.3: Automatic scaling of thread blocks across an arbitrary number of Streaming Multiprocessors, [20].

How to choose the optimal size of a block in order to maximize the performance is not an easy question to answer and it is highly dependent on the particular task and implementation. In practice the size is often chosen by running benchmarks with various sizes to determine the optimal configuration.

In order to make programming 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 calculation involving matrices, 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, that can be added with the two-dimensional thread block code sample in Listing 4.2 is $32 \times 32 = 1024$. 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. Fig. 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 Fig. 4.5. Each level of this hierarchy differs in size, latency and scope. The sizes refers to the amount of bytes that can be stored in the level and the latency refers to the time it takes from requesting data to being able to actually use it. Additionally each level is confined to a specific scope, restricting from where the data can be accessed.

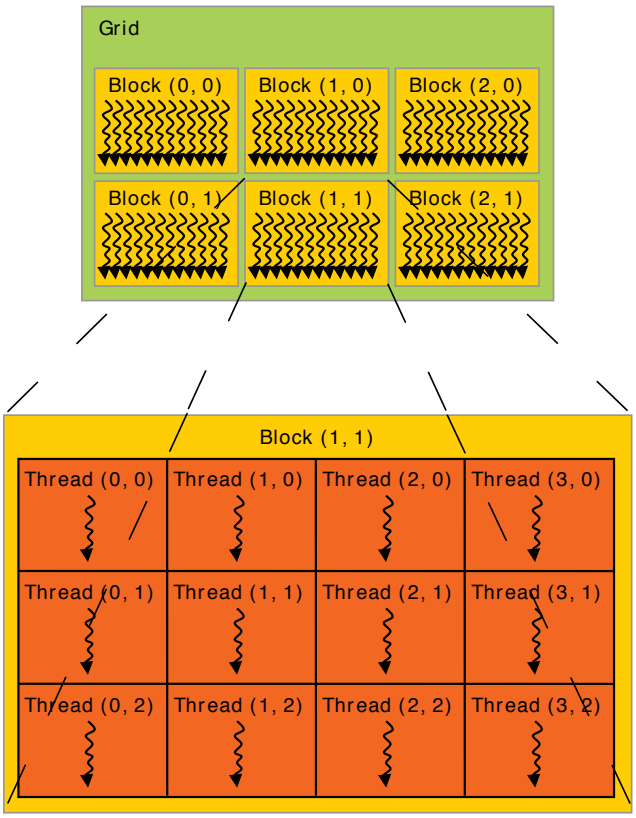


Figure 4.4: CUDA block addressing using 2D grid with 2D thread blocks, [20].

The first level contains the fastest and smallest memory. This is a private memory where each thread has its own area and only this thread can read from and write to 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 $65536 * 32\text{-bit} = 256\text{KB}$ to divide among the threads.

The second level is called shared memory, or sometimes local memory. Although it is slower than the 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 third level contains the largest memory area which is called the global memory. This is the amount of memory used on the packaging of the GPUs to advertise the product. 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 in 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 of the global memory is to make sure that 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.

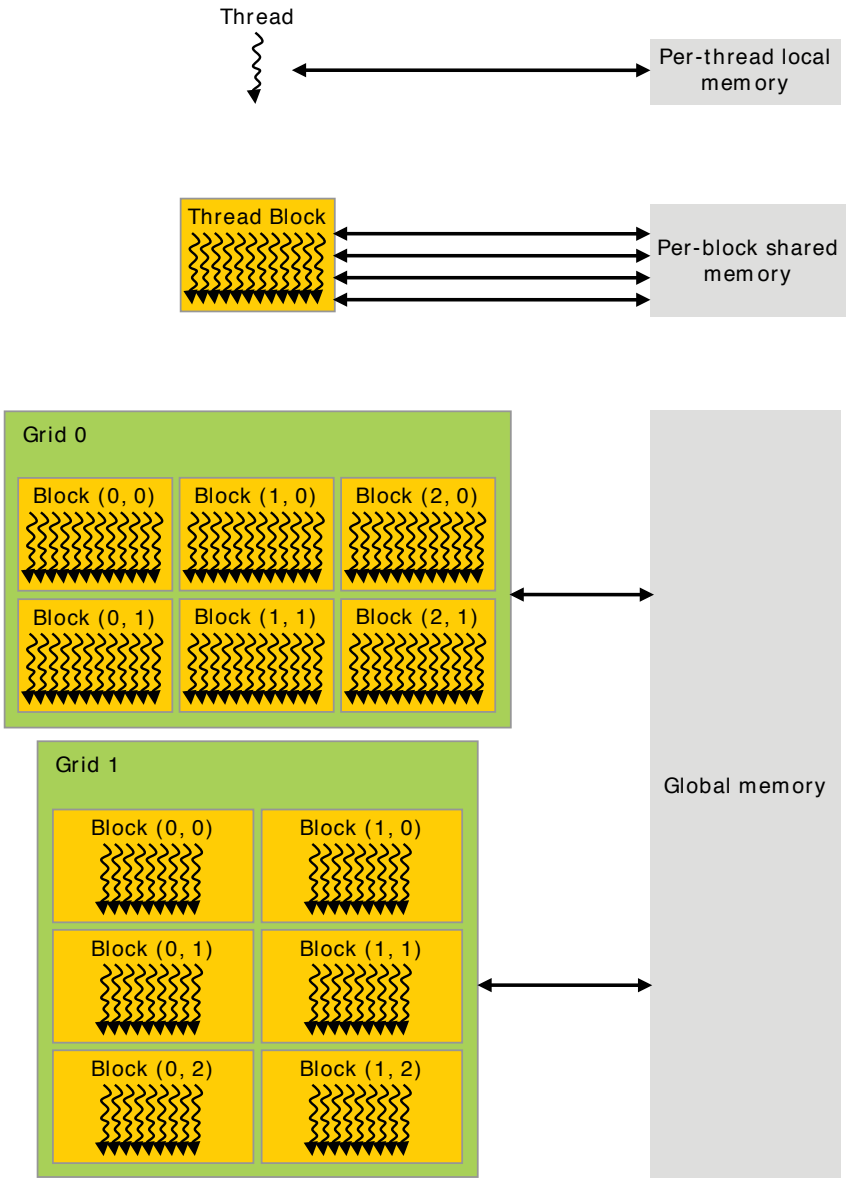


Figure 4.5: CUDA memory hierarchy, [20]. The hiearchy consists of three layers (private, shared, global). Each layer differs in size, latency and scope.

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 is adapted to take advantage of multi-core architectures. The main focus of this thesis is the implementation of the algorithm 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 a discussion of several potential optimization approaches to further improve the performance of the implementation. 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 algorithm developed as part of this thesis is only loosely based on the original serial Fortran implementation. The reason for this is to ensure a clean starting point and to avoid difficulties in adapting the existing code for parallel execution, as it was never intended to be run across multiple cores. This also provides an opportunity to learn from the shortcomings of the old code, and not only parallelize it but also improve the efficiency in general.

The development is done exclusively on a Linux workstation running Ubuntu as this will also be the exact same runtime environment used in later experiments. The build system for compiling and linking the final application is CMake, [14].

It is 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 the CUDA compiler, called *nvcc*, 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*, which allows for a fully automated way to test and validate different algorithm versions against a known correct simulation run. The *validate* mode automatically computes the error and makes debugging of changes easier. The last mode is *benchmark* which executes a particular simulation multiple times and collects and aggregates the 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 different linear solvers. During a simulation, a linear system of equations in the form of Eqn. (3.4) has to be solved in each timestep. This can be one of the most time consuming parts of the algorithm. Hence, the overall simulation time relies heavily on the ability to solve large and dense systems as fast as possible. Therefore, it is very important to weight all the available options carefully when choosing a particular linear solver algorithm or library.

In this work we have compared the performance of a direct solver to certain iterative solvers for solving the system. For the direct solver, the MAGMA library, [15], is used and for the iterative we use the ViennaCL library, [25]. Both libraries will be introduced below.

MAGMA / CuBLAS / OpenBLAS MAGMA is a dense linear algebra library that provides features similar to standard LAPACK functions but for multicore architectures. It also has features to support hybrid algorithms for executing code across multiple GPUs or CPUs at the same time. However, these features are not explored in

this thesis. Instead the focus is 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, [18]. 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, [32], is chosen which is the most up-to-date and high performance library available apart from Intel's Math Kernel Library, [9]. OpenBLAS takes full advantage of multicore systems and is also used for the comparison of our 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 APIs and architectures 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 are 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 that 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* — as described in Sec. 3.6 are run on the GPU.

```
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.

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 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 are described below.

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 Sec. 3.6 and it will also be shown in the benchmark tests presented in Chapter 6.

The goal of the assembly step is to build the matrix and right-hand side for the linear system of equations in the memory. As an example Listing 5.2 shows a pseudocode for the *Assemble System* step. In this example the code is parallelized

```
__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.

using a one-dimensional thread block as described in Sec. 4.3. This means that the code is parallelized for each fiber and that each thread calculates the contributions to this fiber from all the other fibers. Looking at the matrix in Eqn. (3.4) each thread is thus responsible for $3 * N$ rows of the matrix, where N is the total number of terms in the force expansion in Eqn. (3.1). Alongside the one-dimensional thread implementation other options exist. They will be discussed in the optimization Sec. 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 right-hand side elements are computed and used in the next step 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. In the *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 library that implements the linear solver. In case of the direct solver it is the MAGMA library and for the two tested iterative solvers, BiCGStab and GMRES, it is the ViennaCL library. 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 right-hand side and is passed on to the subsequent steps.

Update Velocities By solving the linear system of equations we obtain the coefficients in the force expansion as shown in Sec. 3.1. The *Update Velocities* kernel accumulates the exerted forces for all fibers and updates both the translational and the rotational velocities of the fibers simultaneously by computing the right-hand side in Eqns. (3.2) and (3.3). A two-dimensional thread block version of the kernel is illustrated in Listing 5.3. In a two-dimensional thread block version each individual kernel invocation is responsible for a single pair of fiber interactions. In case of 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 lead to incorrect results for the velocities. Fortunately CUDA provides atomic functions to circumvent this issue which will be discussed in more detail in Sec. 6.2.3.

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, Eqn. (3.6), introduced in Sec. 3.5. As it will be shown during benchmarking in Chapter 6, the required time for this kernel is minuscule compared to the other steps. The kernel scales linearly with the number of fibers and in addition it has a perfectly aligned memory access resulting in close to optimal usage of the GPU hardware.

```
__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.

```
__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.

5.4 Optimizations

During the development of the parallel GPU implementation 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 of the code 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, like the reciprocal of the square root, where also tested and included.

Throughout the optimization phase a 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 implementations, since they showed performance improvements for

both. However, some optimizations and algorithm variations are uniquely suited to the GPU hardware. Below 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

In the original paper, [29], it was observed that the analytical integration of the inner integral in Eqn. (3.5) yielded a performance increase compared to purely numerical integration. Generally, using 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 not 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 Sec. 6.2.1.

5.4.2 Shared memory

As described in Sec. 4.3, the CUDA code is subjected 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, a shared memory version was implemented and tested for the *Assemble System* step, since it is the most performance critical kernel.

To understand the idea, imagine the two-dimensional thread block implementation of the kernel. Then 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 amount of necessary data is small enough to utilize memory caches and that repeated access to global memory can then automatically be avoided. Another factor is the performance characteristic of the kernel. Optimizing for shared memory usage only makes sense if on the one hand 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 to use time efficiently 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 do not have an equivalent fast and comparatively large memory space. Unfortunately, during our testing in Sec. 6.2.2, we did not see a performance advantage by using shared memory for the rigid fiber simulation.

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.

For this thesis we looked at the thread block dimension in particular and how choosing a different approach of parallelizing affects the performance. While doing so, the focus was on the *Assemble System* step. However, the results were also transferred to the *Update Velocities* step, which has a similar structure.

The kernel invocations for the one-dimensional and two-dimensional thread block approach are visualized in Fig. 5.1 using the matrix structure in Eqn. (3.4)

as an example. In the one-dimensional case each invocation is responsible for computing contributions to all sub matrices $\bar{A}_{ml}$. Thus in total this results in M kernel invocations, one for each fiber. For the two-dimensional case each invocation is only responsible for the contribution to a single sub matrix. Here the total number of invocations is $M \times M$. The three-dimensional case 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 dimensions for the system matrix. In the 1D case each thread is responsible for the all interactions. For the 2D case each thread is only responsible for the interaction between a pair of fibers.

The most straight forward approach is to use the one-dimensional thread block which means that the algorithm is parallelized with regard to a single fiber. In this case a single kernel invocation is responsible for multiple rows of the resulting linear system matrix, see Fig. 5.1 (a). 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 as shown in Fig. 5.1 (b). While this decreases the necessary resources, it also requires atomic functions to handle the case when two threads try to update the right-hand side value for the same fiber. By using the so-called atomic functions CUDA streamlines and serializes the memory access. For example the *atomicAdd* function in Listing 5.3 for the *Update Velocities* step accumulates the 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 rigid fibers simulation shows that using a two-dimensional thread block with the associated performance increases, outweigh the potential performance hit of using atomics.

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.

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 show in Sec. 6.2.3, 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 code on the GPU using CUDA. In order to better understand to what degree this goal is achieved, it is crucial to be able to do a fair 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 it does not take advantage of todays multicore CPUs. Furthermore, it is implemented in double precision, which is not suitable for the GPU used in this thesis. 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 parallel CPU code is also implemented. For the parallelization on the CPU the OpenMP library, [2], 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, [32], 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 using a similar approach as the one-dimensional thread block on GPU as discussed in Sec. 5.4.3. This means that each core calculates the interactions for one fiber with all other fibers or put differently each core calculates the entire matrix row belonging to one fiber, as illustrated in Fig. 5.1 (a). As the underlying number of independent threads is much lower on CPUs, different parallelization dimensions did not 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. In the next chapter we will present a number of performance metrics and comparisons between the GPU and CPU implementations.

Chapter 6

Benchmarks

The previous chapter introduced the parallel implementation of the numerical algorithm using NVIDIA's CUDA framework. It presented a practical overview of the GPU implementation and discussed various strategies for optimizing the algorithm.

In this chapter we will present a number of benchmark tests. The main goal of the benchmarks is to measure the performance of the GPU implementation and compare it to the performance of the OpenMP-based CPU implementation. In addition to the comparison between the two different implementations, benchmark tests are also performed to investigate the different approaches for optimizing the code presented in the previous chapter.

6.1 Methodology

The methodology used for the benchmark suite is the same for all presented benchmarks. This ensures comparable results and the fairest comparison possible. A brief overview of the hardware and benchmark scheme used are described in the next sections.

6.1.1 Hardware

All benchmarks are run on the same workstation with specifications listed in Tab. 6.1. These hardware components can be considered a balanced system. This is 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.

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.

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

In order to generate statistically significant and reproducible execution times, all benchmarks tests for both the CPU and GPU implementation uses exactly the same run-scheme.

For each benchmark run we start with a configuration of M fibers randomly distributed in space. In order to exclude configurations where fibers overlap and/or intersect, the random process is modified such that there always is a fixed minimum distance between two fibers. Furthermore, in order to ensure a fair comparison between e.g. different linear solvers, the average distance between fibers are kept fixed for all runs independently of the number of fibers. As will be shown in Sec. 6.3.2, the number of iterations of the iterative solvers depend on the average distance between the fibers. As the average distance become smaller, the condition number of the matrix increases and more iterations are needed for the iterative solver to converge. Thus by keeping the average distance between the fibers fixed, we minimize the influence of the number of iterations in the iterative solver on the execution time.

Using the semi-random fiber configuration, the simulation is run for 10 time steps. To avoid remaining outliers in the configuration potentially causing varia-

tions 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 measured using the final 9 time steps.

To measure how the execution time depends on the number of fibers in the simulation, all tests are run with varying number of fibers starting from 100 fibers up to 2000 fibers using an increment of 100.

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

  int runs = 4;
  while (runs <= MAX_RUNS)
  {
    for(int r = 0; r < runs; ++r)
    {
      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.

In addition, every run is repeated a number of times where each run uses a new random fiber configuration. The final execution time is computed as the average time over the total number of runs. The execution time is measured using the built-in CUDA timing events for the GPU implementation and the Fortran *SYSTEM_CLOCK* function for the CPU implementation.

To further improve the statistical significance of the result the benchmark scheme dynamically adjusts the number of runs performed for each test. If the relative standard error (RSE) of the measured timings collected is too high after a minimum number of runs, more runs are scheduled. This repeats until the relative standard error falls below 20% and reliable timings have been obtained. The

algorithm for producing 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 strategies previously outlined in Sec. 5.4. 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 in Eqn. (3.5). It can be solved either numerically or analytically. Fig. 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, [29], analytical integration is always faster to use than numerical integration.

In contrast to the expected and obtained results using the OpenMP implementation, the CUDA implementation shows a different picture. When we look at the corresponding graph for the CUDA implementation in Fig. 6.2, we observe that the results are reversed. Numerical integration outperforms the analytical integration by a large margin that even increases 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 that 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 continues in lockstep. This issue is referred to as branch divergence and should be avoided as much as possible when writing parallel GPU Code, [19].

To confirm that branch divergence is the reason for the slowdown of the analytic integration version of the GPU implementation 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

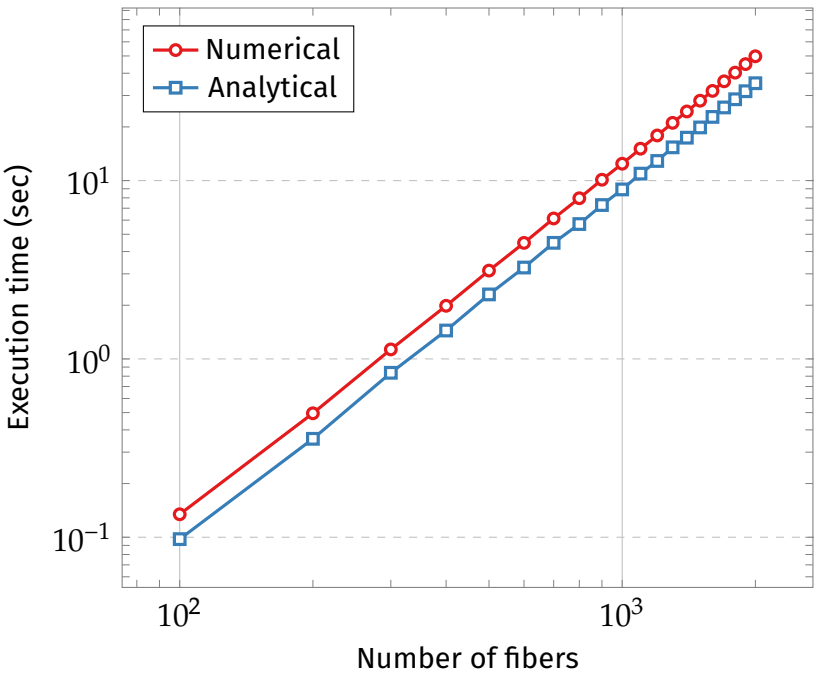


Figure 6.1: Benchmark comparing numerical and analytical integration of the inner integral in Eqn. (3.5) using OpenMP.

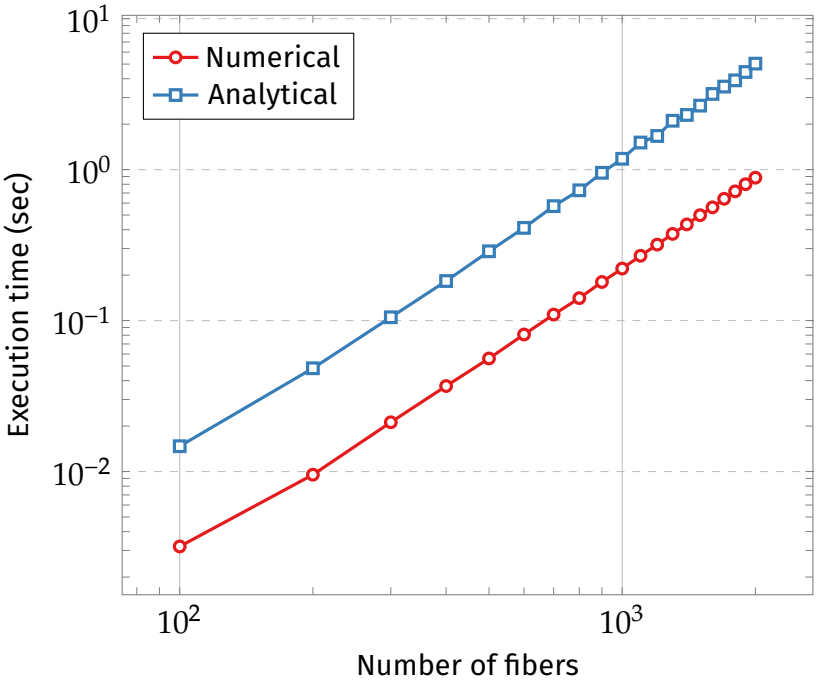


Figure 6.2: Benchmark comparing numerical and analytical integration of the inner integral in Eqn. (3.5) using CUDA.

warp. The metrics for both the numerical and analytical integration can be seen in Tab. 6.2.

Algorithm	warp_execution_efficiency
Numerical	99.01%
Analytical	53.79%

Table 6.2: CUDA performance metric *Warp Exection Efficiency* comparison for the numerical and analytical integration of the inner integral in Eqn. (3.5).

On the one hand the numerical integration is almost 100% efficient, which means that 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 Sec. 5.4.1 the analytical evaluation 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 and explains the decrease in performance on the GPU. The steps involve a simple *if* statement, that switches between two code path depending on how far apart two fibers are. Unfortunately, this workaround is unavoidable to ensure numeric stability.

6.2.2 Shared memory

Since the data transfer between the compute units and the global memory is slow, the second optimization strategy is to try to use shared memory to reduce the amount of data that has to be transferred. For this each Streaming Multiprocessor (SM) has a small amount of locally shared memory. This memory can be accessed from all threads on the SM. If data can be shared, it only has to be transferred 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 described in Sec. 5.4.2, showed no effect on the performance. Even though data can theoretically be shared among different threads it does not result in shorter execution times.

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. evaluating the integrals, takes substantially longer than reading and writing to global memory. This can be explained by looking at the two-dimensional 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. Once the first amount of data has arrived, computations can be performed using it. While the long running computations are executed, other threads can start issuing data loading requests. When the first computation is done and the next set of threads is executed, the data has already been read from memory. As there was no advantage in using shared memory in the compute bound *Assemble System* step, we opted for the simpler implementation without it in all subsequent simulations.

6.2.3 Thread block dimension

The final optimization strategy studied is 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 the usage of potentially slow atomic functions.

The results in Fig. 6.3 indicate that the best option for this particular GPU is 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 when atomic functions are used. Tab. 6.3 lists the *Atomic Transactions* counts for both the two-dimensional and three-dimensional thread block implementation running an example with 2000 fibers. 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.

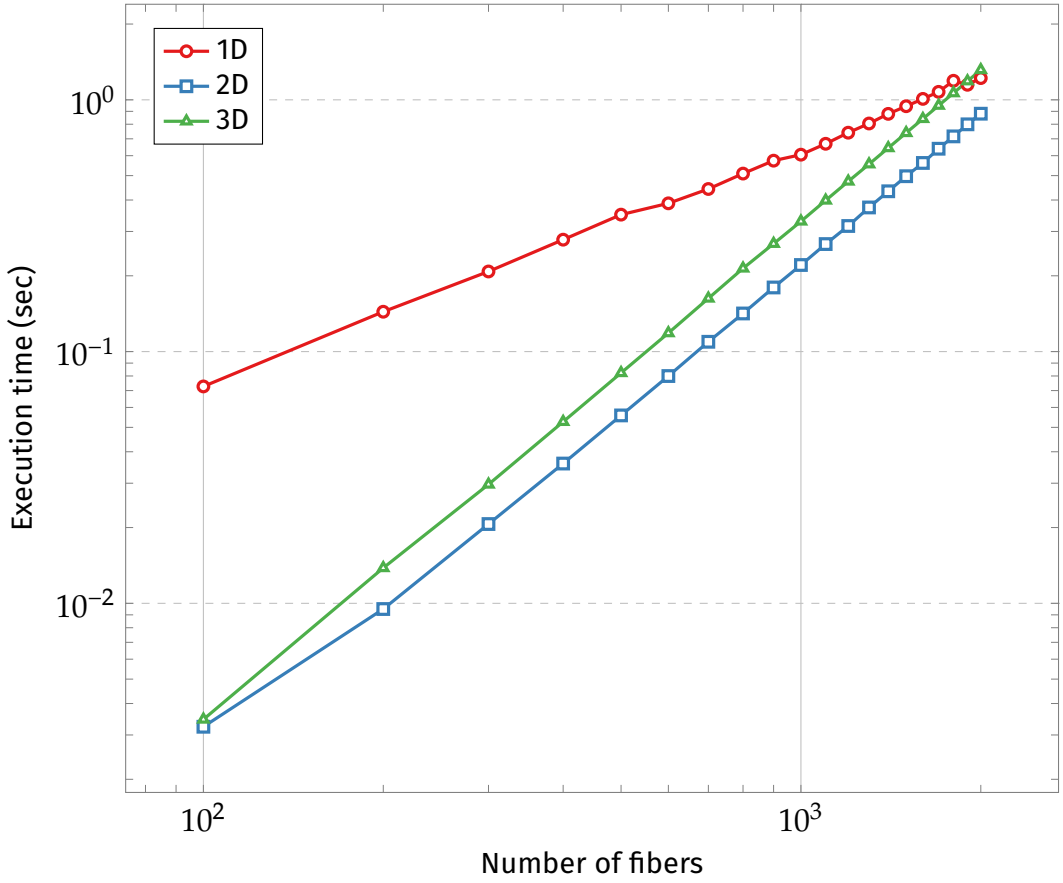


Figure 6.3: Benchmark comparing different parallelization options of the *Assemble System* step using thread block dimensions as described in Sec. 5.4.3.

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

Table 6.3: CUDA performance metric *Atomic transactions* comparison for the 2D and 3D thread block dimensions parallelization of the *Assemble System* step.

Fig. 6.3 also shows that the one-dimensional approach is slower than both the two-dimensional and three-dimensional approach. However, it appears to scale linearly whereas the other two scale quadratically with the number of fibers. 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. 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 Sec. 3.6. It is therefore very important to find the optimal solver in order to arrive at the best performing algorithm overall. We explore both direct and iterative solvers.

6.3.1 Direct solver vs iterative solver on CPU

On the CPU side we use the direct solver provided by the fully parallelized OpenBLAS library. For GMRES we use the single precision Fortran implementation from Frayssé et al., [4], which takes extensive advantage of the underlying BLAS functions parallelized by OpenBLAS.

If we compare the execution time when using GMRES to using a direct solver, we observe that GMRES is faster by a wide margin, see Fig. 6.4. This was also observed in the original serial version of the code, [29].

From an advantage of just $\sim 40\times$ for 1000 fibers this increases to $\sim 300\times$ for the maximum number of 2000 fibers. As expected we can also see in Fig 6.4 that GMRES clearly scales better with the number of fibers than the direct solver.

6.3.2 Fiber concentration effect on GMRES iterations

During the initial benchmark tests we observed large differences in the execution time for different fiber configurations when using GMRES. The reason for this was that for some runs, especially for a high concentration of fibers, GMRES required a large number of iterations to converge.

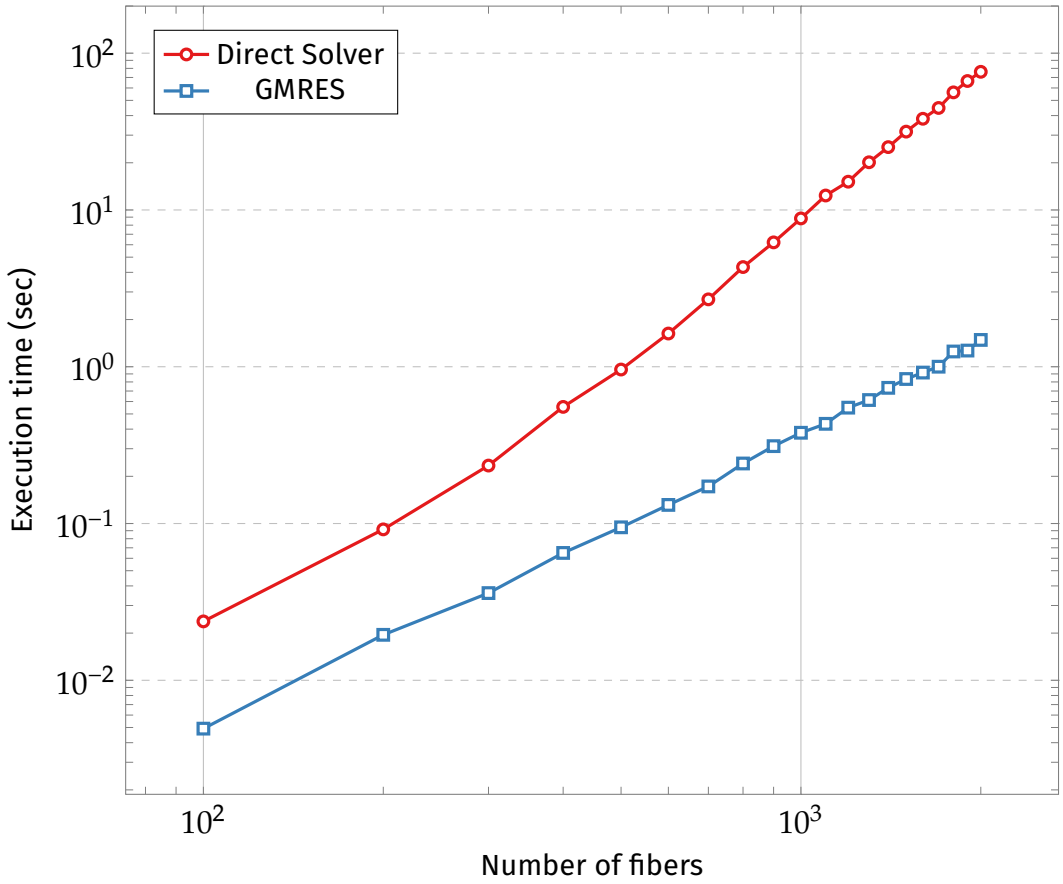


Figure 6.4: Benchmarking comparing linear solvers on the CPU. The direct solver is provided by OpenBLAS, [32], and iterative GMRES solver by Frayssé et al., [4].

When the fiber concentration increases, the average distance between the fibers decreases. To investigate how the number of GMRES iterations depend on the concentration of fibers, we perform a number of runs where the average pair-wise distance between the fibers varies from 0.01 to 40.

In Fig. 6.5 the result is presented. We see that for a decrease in the pair-wise distance, the number of iterations increases quite rapidly. To find the reason for this we have to look at Eqn. (3.5) which forms the basis for the entries in the matrix. When evaluating the integral the Greens function $\mathbf{G}(\mathbf{R})$ defined in Eqn. (2.17) has to be computed. Since $\mathbf{G}(\mathbf{R}) \sim 1/\mathbf{R}$ where $\mathbf{R}$ is given by the distance between fibers, some of the entries in the matrix become very large when the distance is small. The consequence is a growth in the condition number of the matrix and thus an increase in the number of GMRES iterations is required for convergence.

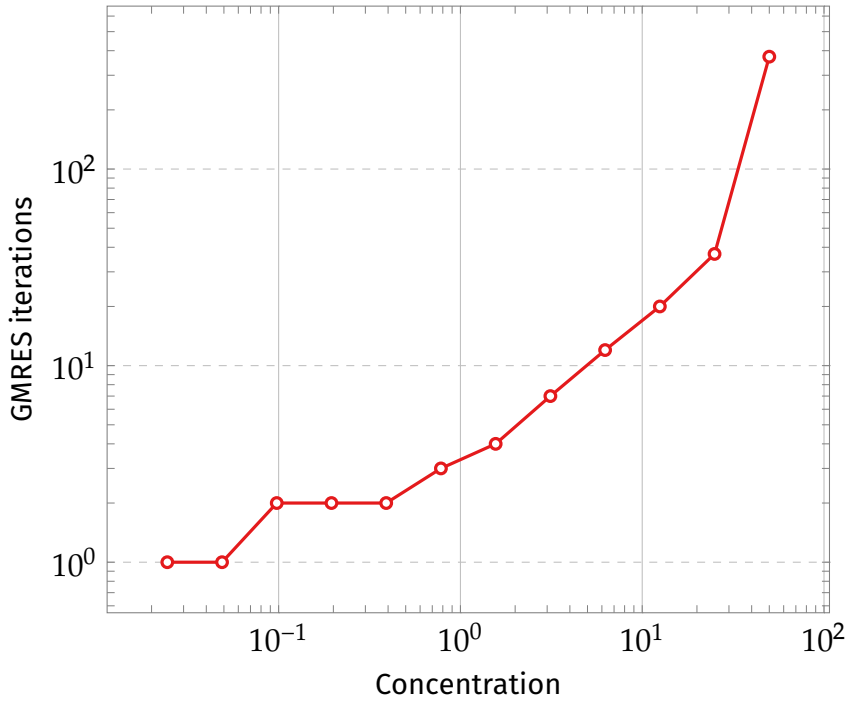


Figure 6.5: Effect of fiber concentration on GMRES iterations. The number of GMRES iterations increases dramatically when the distance between the fibers is small.

This is the reason why we keep the concentration of fibers fixed for all benchmark runs, as described in Sec. 6.1.2.

The result presented above is also important to keep in mind when performing long running fiber simulations. Here, the probability that any two fibers get close to each other is very high. If this is the case, solving the system by GMRES will take longer than expected. In practice it might be more beneficial to switch to the direct solver as it has a predictable runtime. This is especially true using the GPU implementation as the difference in execution time between direct and iterative solver is relatively small. This will be discussed further in the next section.

6.3.3 Direct solver vs. iterative solver on GPU

We will now look at the performance of the direct solver compared to the iterative solvers on the GPU. We use a direct solver provided by the MAGMA library and the iterative solvers, BiCGStab and GMRES, from the ViennaCL library. The benchmark results are illustrated in Fig. 6.6.

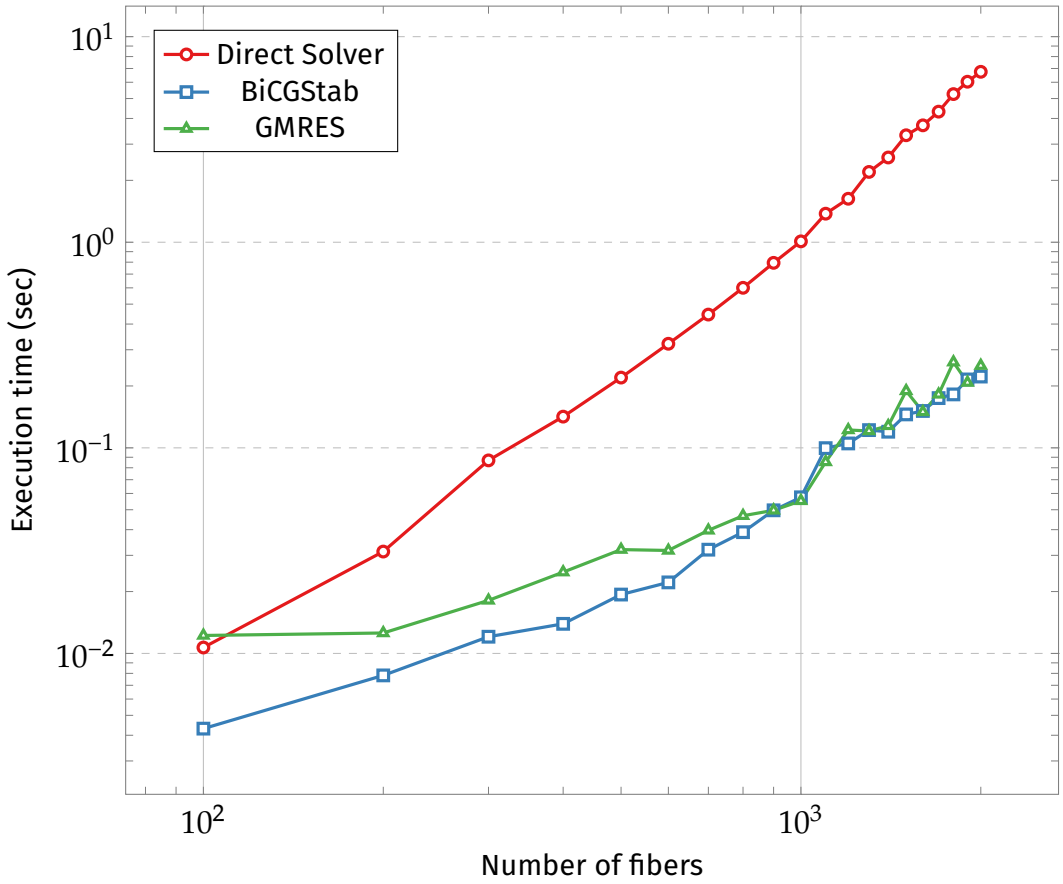


Figure 6.6: Benchmarking comparing linear solvers on the GPU. The direct solver is provided by MAGMA, [15], and iterative solvers, BiCGStab and GMRES, by ViennaCL, [26].

The fact that iterative solvers are faster than a direct solver holds true also for the GPU. However, compared to the results on the CPU 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 $25\times$ faster than the direct solver as compared to the result on the CPU where GMRES was $300\times$ 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 of the algorithm

The next benchmark explores the relative time taken by each step of the algorithm described in Sec. 3.6 and Sec. 5.3. This yields valuable insight into the time allocation of the overall execution time, thereby helping to figure out which steps are the best ones to optimize. The results for the OpenMP implementation can be seen in Fig. 6.7 and for the CUDA implementation in Fig. 6.8. On the CPU the *Assemble System* and *Update Velocities* steps solve the inner integral in Eqn. (3.5) analytically. On the GPU we use a purely numerical approach. The reason for choosing different approaches for evaluating the inner integral is to use the fastest version of the algorithm on both the CPU and GPU, see Sec. 6.2.1.

The results for the OpenMP implementation, Fig. 6.7, 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 Sec. 6.3 this only holds true for sufficiently low concentrations of fibers.

For the CUDA implementation, Fig. 6.8 shows that the *Assemble System* step remains the largest block with 72% of the time. However, now the *Solve System* step takes 20% of the time and *Update Velocites* drops to just 7%. The *Update Fibers* step is also negligible here with below 1%.

This difference between the CPU and GPU implementation 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 $40\times$ 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, [26]. 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 timings for the linear solvers. Here the absolute time for *Solve System* step on the GPU is just ~ 7 seconds com-

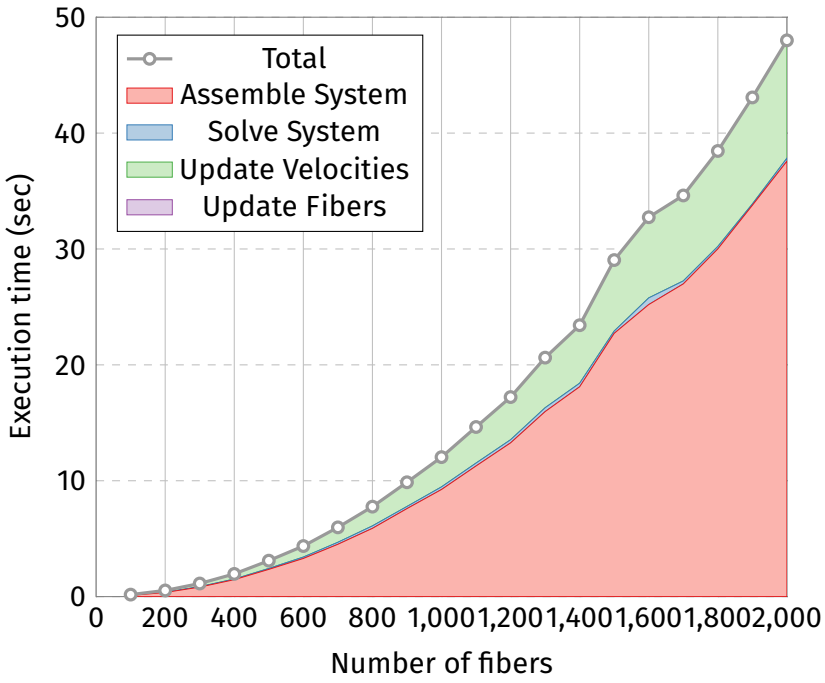


Figure 6.7: Benchmark comparing the execution for each individual step of the algorithm using the OpenMP-based CPU implementation.

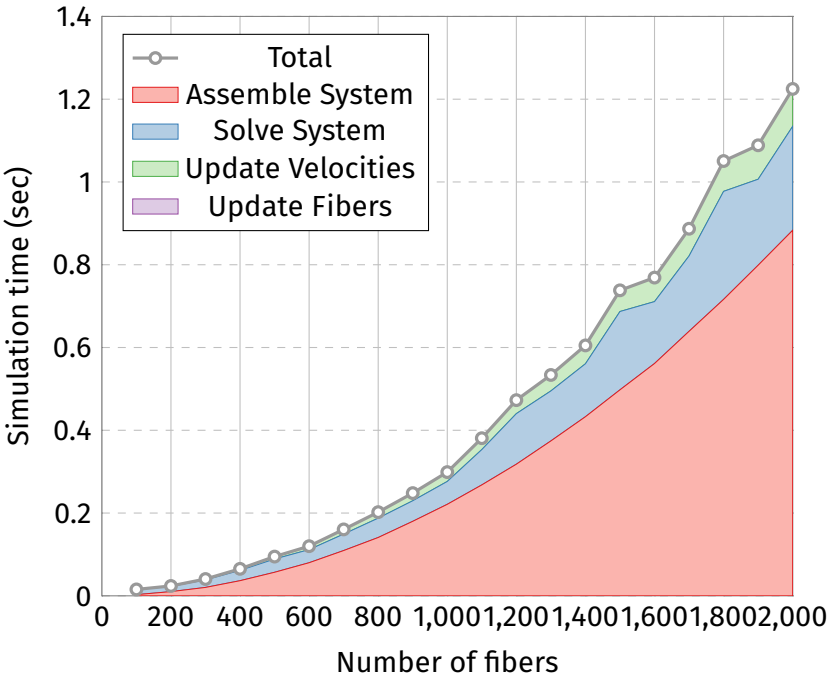


Figure 6.8: Benchmark comparing the execution for each individual step of the algorithm using the CUDA-based GPU implementation.

pared to the ~ 76 seconds on the CPU. Again transferring the relative distribution from the CPU to the GPU yields just a small factor of $\sim 1.4\times$. 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, [6][16]. The underlying architectures of the two approaches are completely different and thus hard to compare. There are approaches where the relative performance is extrapolated from the underlying FLOPs by taking processor count, frequency and memory bandwidth into account, [16]. However, due to intricate hardware details this approach is 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, [12].

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 between the CPU and the GPU performance. In order to come as close as possible given these complexities and constraints, we use a modern CPU and GPU which can be considered a balanced system at the time of writing. Additionally, we implemented a 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 uses the fastest possible version of the algorithms as determined by the performed benchmarks.

The results for the average time required to take a single time step is illustrated in Fig. 6.9. 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 two-dimensional.

The required simulation time on the GPU outperforms the CPU by a wide margin. The GPU version is faster for any number of fibers that we are able to test. The speedup factors for the tested versions are listed in Tab 6.4. CUDA maintains a relative performance of $\sim 40\times$. The only advantage the CPU version has is the

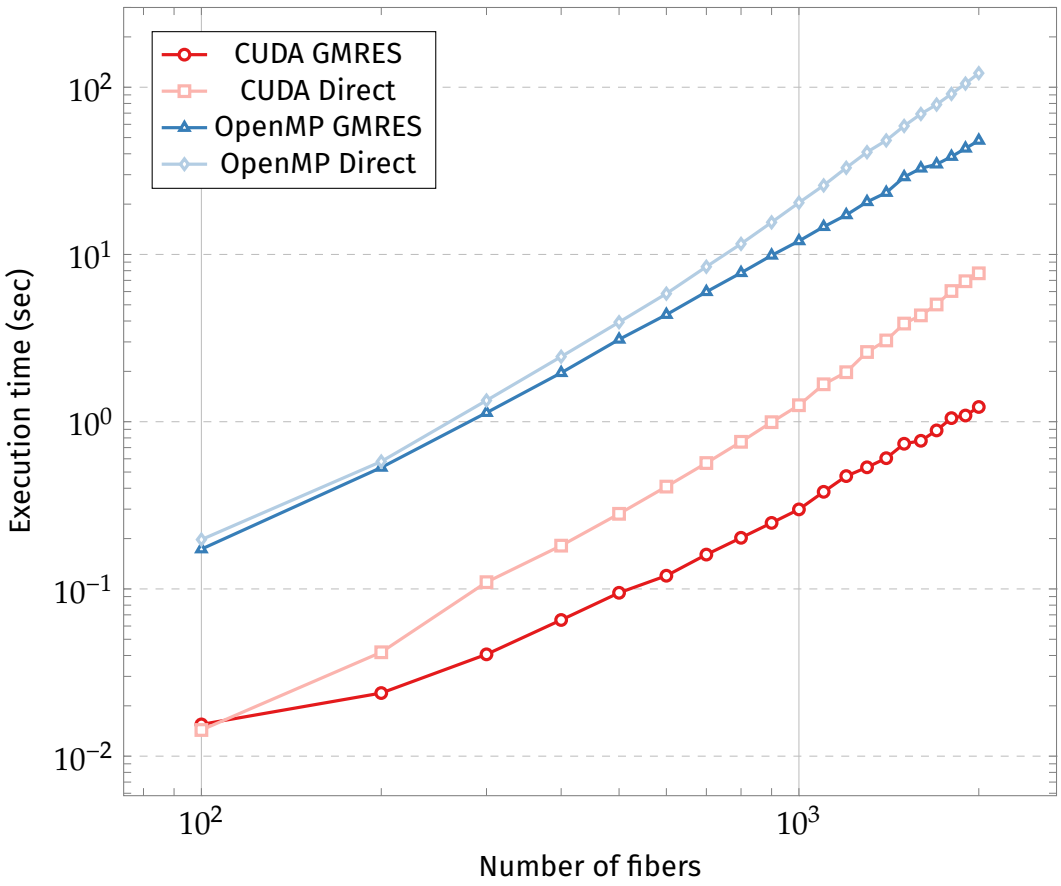


Figure 6.9: Benchmark comparing overall execution time for OpenMP and CUDA. Both the CPU and GPU implementation are run using the fastest algorithm as determined in Sec 6.2 and each implementation is tested with a direct and iterative solver. The CUDA-based GPU implementation outperforms the OpenMP-based CPU implementation by $\sim 40\times$.

potentially larger memory as 4GB on the GPU limits the number of fibers to roughly 2000. So for more fibers than 2000, OpenMP is currently the only option.

M = 2000	OpenMP Direct	OpenMP GMRES	CUDA Direct	CUDA GMRES
OpenMP Direct	$1\times$	—	—	—
OpenMP GMRES	$3\times$	$1\times$	—	—
CUDA Direct	$16\times$	$6\times$	$1\times$	—
CUDA GMRES	$99\times$	$39\times$	$6\times$	$1\times$

Table 6.4: The speedup factors for the overall execution time of a simulation with 2000 fibers for the CPU and GPU implementation. In case of a direct solver the GPU implementation shows a speedup of $16\times$ compared to the CPU implementation. Comparing the two GMRES versions, the CUDA-based GPU implementation outperforms the OpenMP-based CPU implementation by $39\times$.

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 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 $40\times$ is comparable to a difference between a whole day of waiting for the simulation results and a quick 30 minutes result. The saved time also translates directly into the ability to simulate many more fibers than before. Using the original serial implementation the largest simulation possible in a reasonable time frame where around 500 to 800 fibers. Now one single time step with 2000 fibers 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

Numerical experiments

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

In this chapter we will present results from a number of experiments performed to explore the numerical precision of the implementation and to validate the results against prior research. The simulation parameters used in the experiments can be found in the Appendix A.

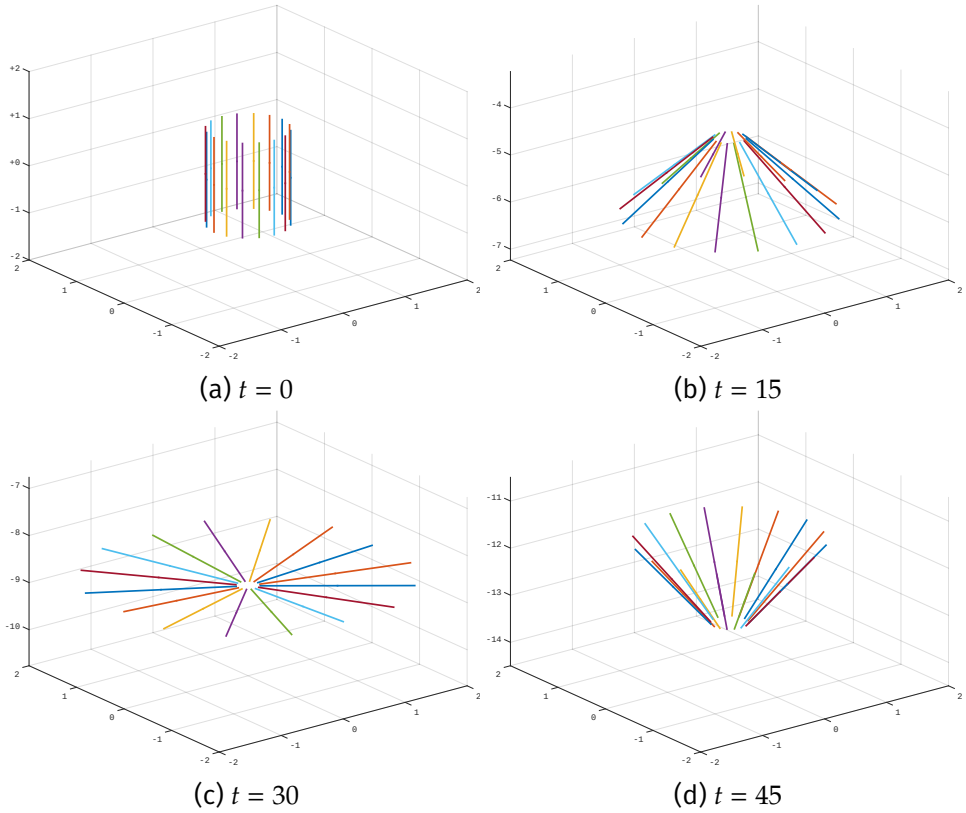
We begin by looking at tumbling orbits, a very delicate experiment requiring good numerical precision. Next we perform numerical experiments of a very interesting physical phenomenon that arises when a spherical cloud of fibers sediments. The results are validated against similar experiments performed by others. Furthermore, we will have a brief exploration of the effects of the number of fibers and the concentration of fibers on the spherical cloud simulation. All experiments are run with the new GPU algorithm implemented in the thesis. The linear system is solved using the direct solver from MAGMA in order to avoid variations in the run time.

7.1 Tumbling orbits

To verify that the single precision GPU implementation is able to replicate the result obtained with the original double precision code, we perform a very simple experiment where a small number of fibers are set up with perfectly symmetrical positions and orientations. Initially, all fibers are evenly distributed on a circle and aligned vertically with gravity. During the simulation, while sedimenting, the

fibers begin to rotate from their vertical orientation towards a horizontal position. Afterwards, they continue rotating back into the vertical position. This motion is referred to as a tumbling motion and as long as there are no disturbances or numerical precision issues it repeats forever.

This simple but very interesting problem has also been simulated by Gustavsson and Tornberg, [8]. Additionally an even more simplified version with only two fibers was studied both numerically and experimentally by Jung et al., [11]. This example is thus ideally suited to test and verify the numerical precision of the GPU implementation. A visualization of the result using the GPU code for 16 fibers evenly distributed around a circle with a radius of 0.55, is shown in Fig. 7.1.



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

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

Figure 7.1: Visualization of tumbling orbits. Small number of perfectly symmetrically distributed fibers around a circle are allowed to sediment due to gravity. The fibers perform a periodical motion alternating between a vertical and horizontal orientation in the direction to gravity.

Initially, the fibers are aligned vertically and are sedimenting with the maximum velocity. As they rotate into the horizontal orientation the velocity decreases 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.

Fig. 7.2 shows a graph of the obtained sedimentation velocity of a single fiber over time using 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. Therefore it is sufficient to study the sedimentation velocity of one fiber only. 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. The difference in velocity between the two implementations is of the order 10^{-4}. Hence, even for this delicate experiment, where a small disturbance will cause a deviation from the periodic orbit, the single precision accuracy is sufficient.

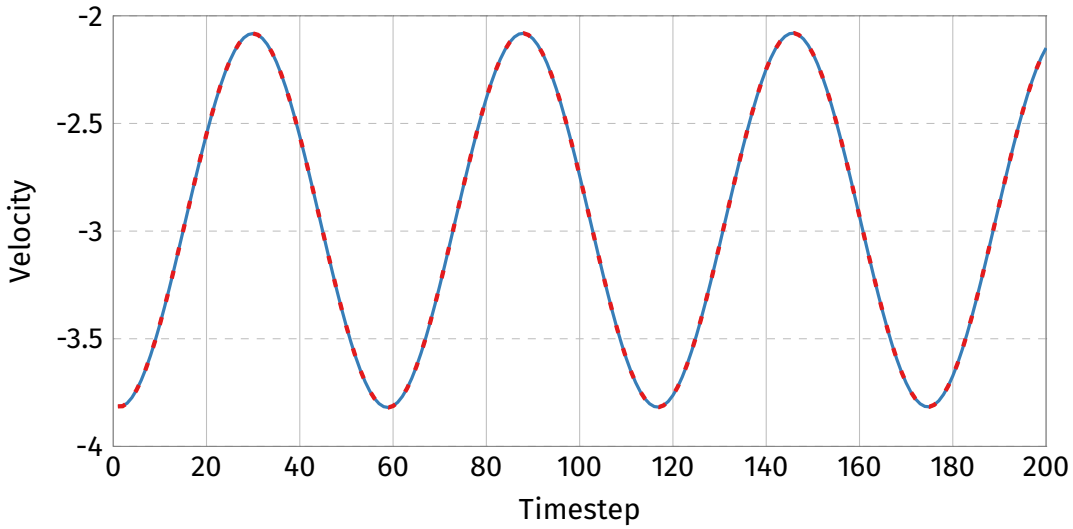


Figure 7.2: Comparison of sedimentation velocity for single precision (solid line) and double precision (dashed line) simulation.

7.2 Sedimenting of a spherical cloud

The next example is a more chaotic system with a large number of interacting fibers. In the numerical experiment 2000 fibers are initially distributed in the form of a spherical cloud. Both their positions and orientations are random inside the cloud. Due to gravity the cloud will sediment. This experiment has been studied

in several papers, e.g. [1][17][22]. 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.

In Fig. 7.3 we can see how the interacting fibers, beginning from the spherical shape, slowly start 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 somewhat resembles the result for the tumbling orbits presented in Sec. 7.1.

After some time the torus breaks and the fibers are split into multiple smaller cloudlets, which continue to sediment separately and slowly form their own smaller torus. However, due to tiny variations these tori can be harder to see.

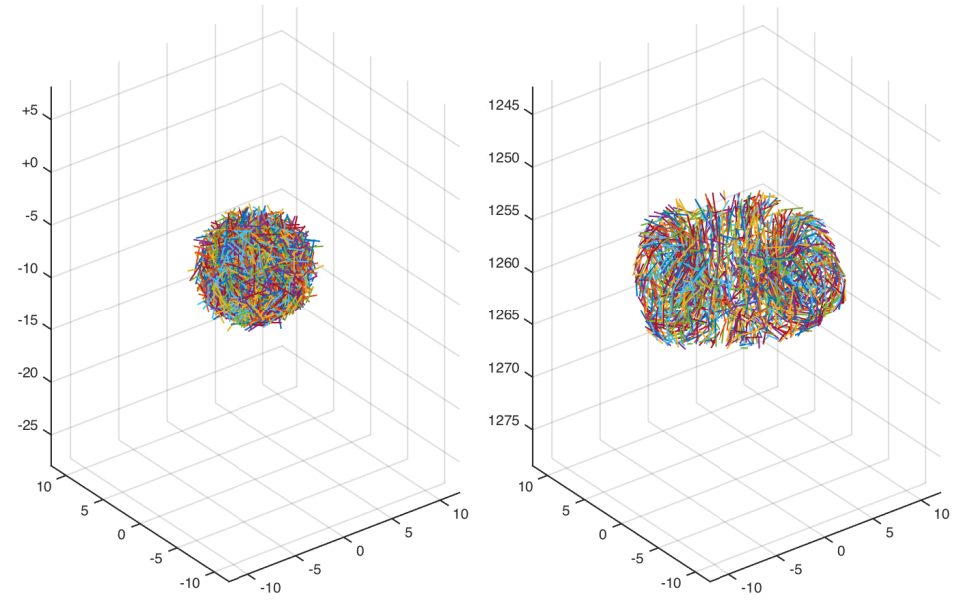
The simulation of the presented example only takes ~ 8 seconds per timestep using the GPU implementation. Consequently it is possible to perform the 500 time steps of the simulation in a little bit over 1 hour. Simulating this many fibers in such a short time will allow for new research of this interesting phenomenon. One interesting question is to determine what influences the stability of the torus that is, when time does the torus break up.

7.2.1 Spherical cloud break-up

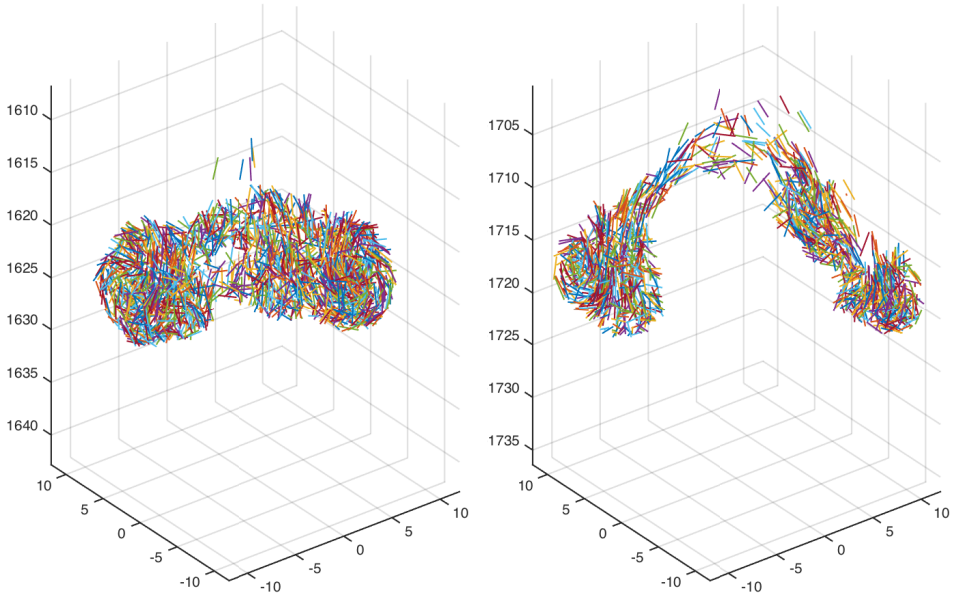
One interesting question is how to determine what influences the stability of the torus that is, when in time does the torus break-up.

How to determine the exact break up time is quite challenging. This problem was also examined by Park et al., [22]. In [22] the authors define the break-up time as the instant when the torus starts to bend prior to actually breaking. However, they do not present a precise definition of the bending in terms of measurable quantities. In order to systematically detect the time when the torus breaks we have developed a measure of the break-up using the standard deviation of the vertical radius of the torus. We will use the same setup as above to exemplify how we define this measure.

First we need to define which of the fibers belong to the torus. We use the same definition as described in [22]. Let R_0 denote the radius of the initial cloud. As the cloud sediments and starts forming a torus, there is a small leakage of fibers in its vertical tail. To find these fibers, such that they can be removed from the active set of fibers defining the torus, we remove all fibers with a vertical distance from the center of mass of the torus larger than R_0. In Fig 7.5 (b) we can see how the



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



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

Figure 7.3: Visualization of sedimenting spherical cloud. Fibers are randomly distributed in the form of a spherical cloud and sediment due to gravity. The fibers proceed to form a turning torus. Eventually the torus breaks up into subclusters.

number of fibers in the torus decreases with time. Using the remaining fibers, we compute the horizontal radii R_x and R_y of the torus by computing the maximum distance from the center of mass to the center of the fibers. In the same way we compute the vertical radius R_z. The torus shape is illustrated in Fig. 7.4 where we display the three different radii, R_x, R_y and R_z.

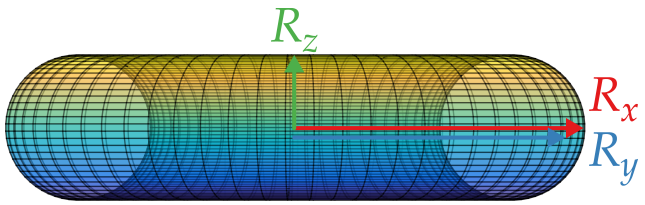
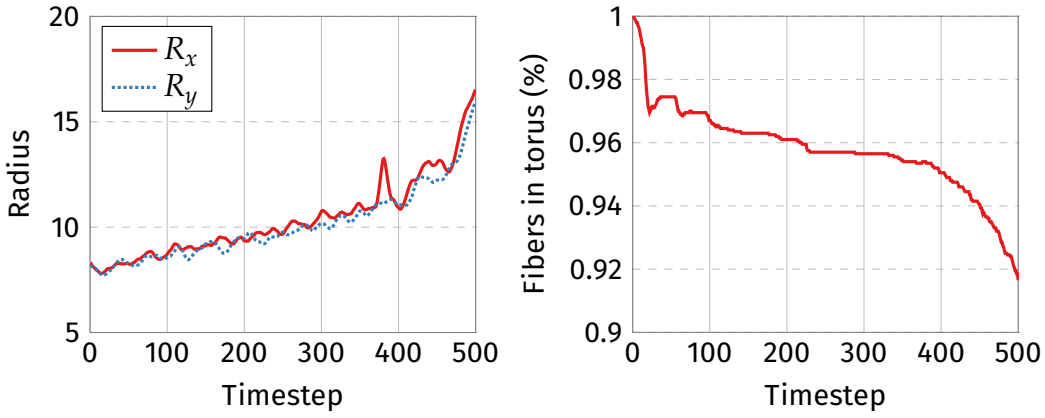


Figure 7.4: Torus Shape. The torus is defined by the two horizontal radii R_x and R_y and the vertical radius in the direction of gravity R_z.

In Fig. 7.5 (a) the radii in the horizontal direction is presented as a function of time. We can see how the radii increase at a steady rate until break-up. At the time of break-up there is a more sudden increase. If we instead look at the time evolution of the standard deviation of the radius in the vertical direction, R_z^{std}, we observe that it continuously decreases until it reaches a minimum value after which it starts to increase again, see Fig. 7.5 (c). The time at which the minimum value is reached can be identified with the break-up of the torus. We have verified this finding manually against many simulations and it showed a great agreement with the visually detected time for break-up. Using the minimum value of R_z^{std} as a detector for the break-up time is preferable compared to using e.g. the sudden increase in the horizontal radii since it is easier to automatically detect a minimum value than a sudden increase.

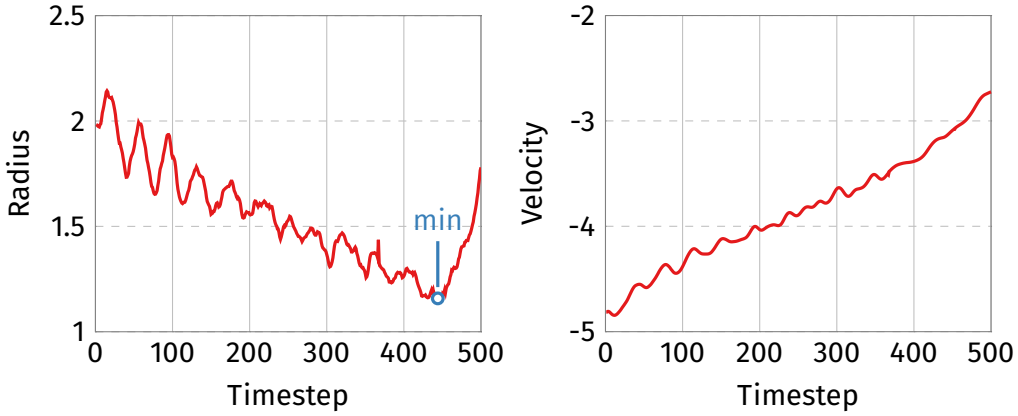
A surprising result is that the standard deviation of R_z oscillates with time. This implies that the torus is periodically expanding and shrinking until it breaks. The same periodic behavior can also be seen in the sedimentation velocity, V_z, of the torus, see Fig. 7.5 (d). Here, we define the sedimentation velocity as the average value of the vertical velocity of all fibers in the torus. The periodic change in the velocity is similar to the motion observed in the simple tumbling orbits experiment presented in Sec. 7.1. Hence, it seems like the turning torus with 2000 fibers is a more chaotic version of the tumbling orbits.

The results presented in Fig. 7.5 are comparable to results presented in Park et al. [22]. Using our definition of break-up time we will now investigate the effect of the number of as well as the concentration of fibers on the stability of the torus.



(a) Torus horizontal radius.

(b) Percentage of remaining fibers.



(c) Torus Radius$_z$ Standard Deviation.

(d) Mean sedimentation velocity.

Figure 7.5: Time evolution of different quantities related to the sedimenting torus. In the beginning the torus loses some fibers and the horizontal radii R_x and R_y increase slowly. The vertical radius R_z undergoes a periodical contraction and expansion, which can also be seen in the peridocal change in the sedimentation velocity V_z. The time of the break-up is defined as the minimum value of the standard deviation of the vertical radius R_z^{std}.

7.2.2 Fiber concentration effect on break-up time

The first parameter to explore is the concentration of fibers, which we define in terms of the average distance between a fiber and its closest neighbor. In the numerical experiments, we fixed the number of fibers at 2000 and varied the concentration in the spherical cloud by varying its initial radius, R_0. The break-up time is estimated using the measure defined in Sec. 7.2.1.

In Fig. 7.6 the break-up time is presented as a function of the concentration of fibers. We can see that there is a clear correlation. When the concentration decreases, the time until the torus breaks up increases. A possible explanation to this is that when the cloud sediments and starts forming a torus, a large rotating velocity field is created in the fluid. The strength of the velocity field depends on how strong the interaction between the fibers is. When the fibers are far apart the interaction between the fibers is small and the rotating motion will not be as strong, leading to a longer break-up time.

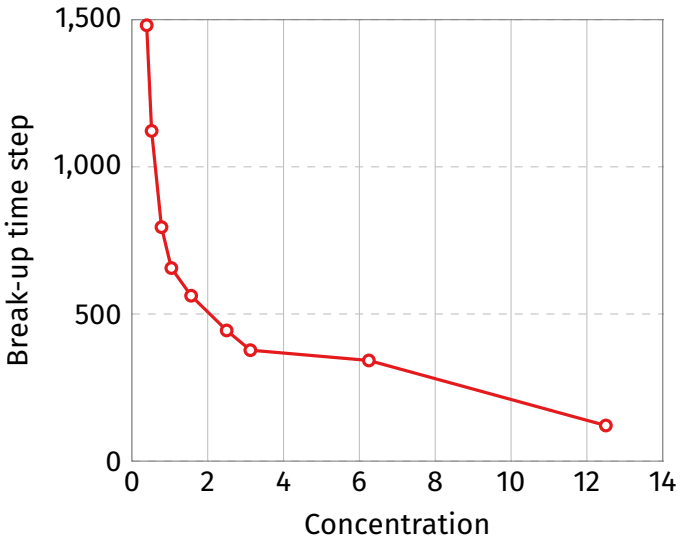


Figure 7.6: Effect of fiber concentration on torus break up time.

7.2.3 Number of fibers effect on break-up time

The second parameter we studied was the number of fibers initially included in the cloud. In this set of experiments, the concentration is fixed by keeping the average distance between the fibers fixed to 0.4. We measure the break up time as we vary the number of fibers from 100 to 2000. In Fig. 7.7 the result is displayed.

Here we find that there is a positive correlation between the number of fibers and the break-up time, however not as strong as in the case when we varied the concentration. These results also match the results found by Park et al., [22].

The results presented in Sec. 7.2.2 and Sec. 7.2.3 show that both fiber concentration and the number of fibers affect the break-up time of the torus. Future research should explore these correlations closer and include other studies such as the dependence of the initial shape of the cloud on the process.

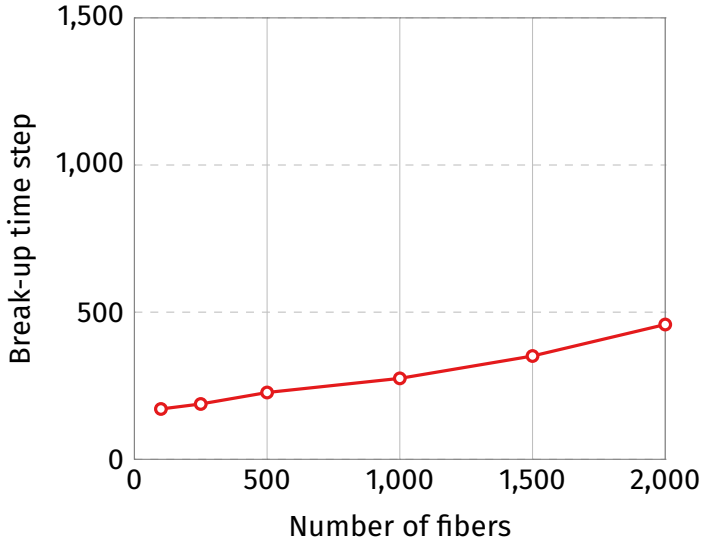


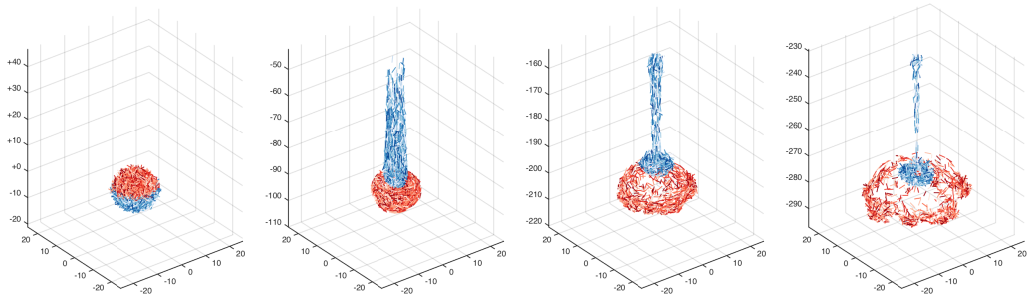
Figure 7.7: Effect of number of fibers on torus break up time. Here the concentration of fibers is constant.

7.3 Cloud of fibers with different densities

The final experiment resembles a setup investigated by Bülow et al., [1], where the authors studied the settling behavior of a spherical cloud with particles of different size. Here, we will study the sedimentation of a spherical cloud with fibers of different densities. We divide the total number of fibers into two different density groups and arrange them inside a spherical cloud, which sediments due to gravity. To separate the two groups visually, they are colored in red and blue. The blue group has the lower density and the red group the higher density. The ratio between the densities is set to blue/red $= 0.75/1.0$.

Fig. 7.8 shows the results from a simulation at different timesteps, where the two density groups are initially separated into the top and bottom half of the

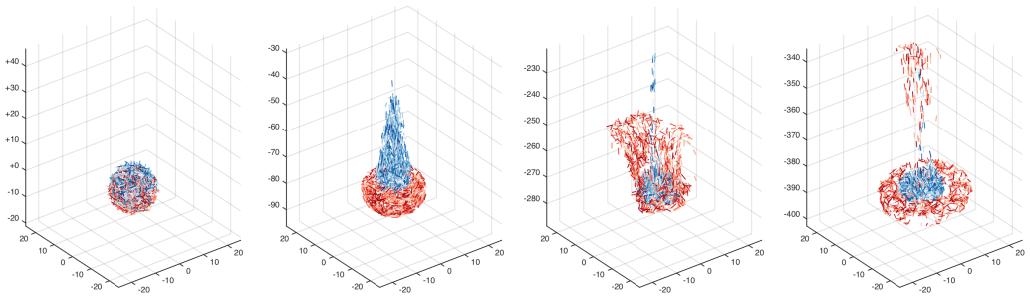
spherical cloud. Since the fibers are clearly separated, this particular configuration is referred to as an unmixed cloud. Initially the fibers with the higher density are in the upper half of the cloud and the fibers with the lower density are in the bottom half, see Fig. 7.8 (a). After a short time, we can see in Fig. 7.8 (b) that the high density fibers have almost completely dropped through the low density fibers, leaving a large trail of low density fibers behind. After a while, the high and low density fibers are almost totally separated and proceed independently of each other to form the characteristic torus, see Fig. 7.8 (c). As the torus formed by the high density fibers grows in size and becomes less dense its sedimentation velocity will decrease. As depicted in Fig. 7.8 (d) the low density group of fibers will eventually catch up with the high density fibers and drop through the larger torus. The tori are not very stable and soon afterward they will break up.



(a) $t = 0$ (b) $t = 20$ (c) $t = 60$ (d) $t = 100$

Figure 7.8: Unmixed cloud. Initially the high density fibers (red) are in the top half of the cloud and the low density fibers (blue) are in the bottom half.

Next we perform a similar experiment but with the fibers of different densities randomly distributed inside the cloud, see Fig. 7.9 (a). We refer to this setup as a mixed cloud. The beginning of the simulation behaves similar to the unmixed case, but the separation does not happen as fast and far fewer low density fibers are lost in the tail of the cloud, Fig. 7.9 (b). Once again two tori are formed, but in contrast to the unmixed case, both groups of fibers stay closer together, which prevents the fibers of different densities from completely separating. In Fig. 7.9 (c) we can see how the high density torus is sucked down by the passing low density torus. During the following timesteps the two tori appear to form an interlocked torus, which continues to sediment, Fig. 7.9 (d). Eventually the interlocked torus breaks up and fibers scatter.



(a) $t = 0$ (b) $t = 20$ (c) $t = 100$ (d) $t = 150$

Figure 7.9: Mixed cloud. Initially the high (red) and low (blue) density fibers are randomly mixed inside the cloud.

The results displayed for the mixed and unmixed cloud with rigid fibers do agree on a qualitative level with the results presented in Bülow et al., [1]. However, both simulation methods and setups are very different which makes a quantitive comparison very challenging. Furthermore our experiment are only in an initial stage and many more experiments have to be carried out to get a deeper understanding of the behavior of rigid fibers of different densities in a spherical cloud. Exploring the impact of initial fiber configuration, the density ratio and the fiber concentration are all exciting future research opportunities.

Chapter 8

Conclusion

In this thesis we have developed a completely new parallel GPU implementation, using CUDA, for the numerical simulation of rigid fibers suspension. The new GPU implementation of the numerical algorithm outperforms the older CPU version with a speed up of $20\times$ to $40\times$. This enables simulations with many more fibers than before. 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 developed an algorithm suitable for taking advantage of massively parallel computational power available in modern GPUs. We outline the required steps to implement the algorithm using CUDA and investigate a number of different optimization strategies to further improve the performance. Both versions of the algorithm, GPU specific implementation details and the choice of linear solver are investigated. To reach the goal of comparing the performance between the CPU and GPU implementation the new parallel algorithm is backported to the CPU using OpenMP. Thereby improving the fairness of the comparison as much as possible.

Using extensive benchmarking we show that different versions of the two implementations perform differently depending on the underlying architecture. E.g. optimizations developed for the CPU implementation actually slows 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.

To test and validate our new parallel implementation, we perform a number of simulations of known experiments for sedimenting fibers. Using the test-case of tumbling orbits we show that the numerical precision of our single precision implementation is able to reproduce the result obtained from the former double precision Fortran simulations. As an example of the ability to handle many fibers, simulations with a sedimenting spherical cloud of fiber are performed. This setup displays a very interesting physical behavior. Once the fibers start to sediment, the spherical cloud is transformed into a torus that eventually breaks up into smaller cloudlets. Our results show an excellent agreement with both experimental and numerical work performed previously. Additionally, we show that the stability of the torus, that is the time until break-up, is positively correlated with the increasing distance between fibers as well as the total number of fibers. Finally, we look at the sedimenting behavior of a mixed density spherical cloud of fibers. In contrast to the uniform density spherical cloud the two groups of different density fibers separate and form independent tori. These tori undergo complex interactions and eventually break-up. Further research in this area is helped greatly by the fact that new cases 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 the future we would like to explore how to get around this restriction. One possible direction is to avoid storing the matrix completely. Instead the required computation will be carried out iteratively and on-demand. Concerning the memory, this will allow for an unlimited number of fibers, but coming at the cost of a largely increased computational time. If this trade off is worth it remains to be seen.

Another exciting possibility is to use 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 efficiency it might be worth looking at the numerical algorithm itself. Adapting a different algorithm based on a 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] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[3] X. J. Fan, N. Phan-Thien, and R. Zheng. “A direct simulation of fibre suspensions”. In: *Journal of Non-Newtonian Fluid Mechanics* 74.1998 (1998), pp. 113–135.

[4] V. Frayssé, L. Giraud, S. Gratton, and J. Langou. “A set of GMRES routines for real and complex arithmetics”. In: *CERFACS, Toulouse Cedex, France, Tech. Rep. TR/PA/97/49* (1997).

[5] T. Götz. “Interactions of Fibers and Flow: Asymptotics, Theory and Numerics”. PhD thesis. 2000, p. 117.

[6] C. Gregg and K. Hazelwood. “Where is the data? Why you cannot debate CPU vs. GPU performance without the answer”. In: *ISPASS 2011 - IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Apr. 2011, pp. 134–144.

[7] É. Guazzelli and J. Hinch. “Fluctuations and instability in sedimentation”. In: *Annual Review of Fluid Mechanics* 43.1 (Jan. 2010), pp. 97–116.

[8] K. Gustavsson and A. K. Tornberg. “Gravity induced sedimentation of slender fibers”. In: *Physics of Fluids* 21.12 (2009), pp. 1–15.

[9] Intel Corporation. *Intel Math Kernel Library*. 2014. URL: `https://software.intel.com/en-us/intel-mkl`.

[10] C.G. Joung, N. Phan-Thien, and X. J. Fan. “Direct simulation of flexible fibers”. In: *Journal of Non-Newtonian Fluid Mechanics* 99.February 2001 (2001), pp. 1–36.

[11] S. Jung, S. E. Spagnolie, K. Parikh, M. Shelley, and A. K. Tornberg. “Periodic sedimentation in a Stokesian fluid”. In: *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics* 74.3 (Sept. 2006).

[12] S. Kamil, J. Shalf, and E. Strohmaier. “Power efficiency in high performance computing”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008), pp. 1–8.

[13] Khronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*. 2014. URL: https://www.khronos.org/opencl/.

[14] Kitware. *CMake*. 2014. URL: http://www.cmake.org/.

[15] Innovative Computing Laboratory. *MAGMA: MAGMA User Guide*. 2014. URL: http://icl.cs.utk.edu/projectsfiles/magma/doxygen/.

[16] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. “Debunking the 100X GPU vs. CPU myth”. In: *ACM SIGARCH Computer Architecture News* 38 (2010), p. 451.

[17] B. Metzger, M. Nicolas, and É. Guazzelli. “Falling clouds of particles in viscous fluids”. In: *Journal of Fluid Mechanics* 580 (May 2007), p. 283.

[18] NVIDIA Corporation. *CuBLAS*. 2014. URL: https://developer.nvidia.com/cuBLAS.

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

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

[21] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. “GPU Computing”. In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899.

[22] J. Park, B. Metzger, É. Guazzelli, and J. E. Butler. “A cloud of rigid fibres sedimenting in a viscous fluid”. In: *Journal of Fluid Mechanics* 648 (Apr. 2010), p. 351.

[23] C. S. Peskin. “The immersed boundary method”. In: *Acta Numerica* 11 (2002).

[24] C. Pozrikidis. *Boundary Integral and Singularity Methods for Linearized Viscous Flow*. 1992.

[25] K. Rupp. *ViennaCL - The Vienna Computing Library*. 2014. URL: http://viennacl.sourceforge.net/doc.

[26] 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.

[27] P. Skjetne, R. F. Ross, and D. J. Klingenberg. "Simulation of single fiber dynamics". In: *Journal of Chemical Physics* 107 (1997), pp. 2108–2121.

[28] J. M. Stockie and S. I. Green. "Simulating the Motion of Flexible Pulp Fibres Using the Immersed Boundary Method". In: *Journal of Computational Physics* 147 (1998), pp. 147–165.

[29] A. K. Tornberg and K. Gustavsson. "A numerical method for simulations of rigid fiber suspensions". In: *Journal of Computational Physics* 215.1 (June 2006), pp. 172–196.

[30] A. K. Tornberg and M. J. Shelley. "Simulating the dynamics and interactions of flexible fibers in Stokes flows". In: *Journal of Computational Physics* 196.1 (May 2004), pp. 8–40.

[31] S. Yamamoto and T. Matsuoka. "Dynamic simulation of fiber suspensions in shear flow". In: *Journal of Chemical Physics* 102.5 (1995), pp. 2254–2260.

[32] X. Zhang. *OpenBLAS*. 2014. URL: http://www.openblas.net/.

Appendix A

Simulation Parameters

Parameters for GMRES iterations experiment in Sec 6.3.2.

Parameter	Value
Number of fibers	2000
Average distance	0.02–40.96
Timestep	0.1
Slenderness	0.01
Number of terms in force expansion	5
Number of quadrature intervals	8
Number of quadrature points per interval	3

Parameters for tumbling orbits experiment in Sec 7.1.

Parameter	Value
Number of fibers	16
Average distance	0.2146
Timestep	0.1
Slenderness	0.01
Number of terms in force expansion	5
Number of quadrature intervals	8
Number of quadrature points per interval	3

Parameters for sedimenting sphere experiment in Sec 7.2.

Parameter	Value
Number of fibers	2000
Average distance	0.2
Timestep	0.1
Slenderness	0.01
Number of terms in force expansion	5
Number of quadrature intervals	8
Number of quadrature points per interval	3

Parameters for fiber concentration effect on break-up in Sec 7.2.2.

Parameter	Value
Number of fibers	2000
Average distance	0.08–2.56
Timestep	0.1
Slenderness	0.01
Number of terms in force expansion	5
Number of quadrature intervals	8
Number of quadrature points per interval	3

Parameters for number of fibers effect on break-up in Sec 7.2.3.

Parameter	Value
Number of fibers	100–2000
Average Distance	0.4
Timestep	0.1
Slenderness	0.01
Number of terms in force expansion	5
Number of quadrature intervals	8
Number of quadrature points per interval	3

Parameters for un/mixed sphere in Sec 7.3.

Parameter	Value
Number of fibers	2000
Average Distance	0.4
Timestep	0.1
Slenderness	0.01
Number of terms in force expansion	5
Number of quadrature intervals	8
Number of quadrature points per interval	3