



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

*Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
 Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis.
 Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasel-
 lus massa massa, auctor rutrum, vehicula ut, porttitor a,
 massa. Pellentesque fringilla. Duis nibh risus, venenatis
 ac, tempor sed, vestibulum at, tellus. Class aptent tac-
 iti sociosqu ad litora torquent per conubia nostra, per
 inceptos hymenaeos.*

Referat

GPU simulering av stela fibrer

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis. Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasel-lus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti so-ciosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Contents

1	Introduction	1
2	Theoretical Foundation	3
2.1	Stokes flow and Stokeslet	3
2.2	Boundary integral formulation	5
2.3	Slender fibers	7
2.3.1	Nondimensionalization	7
2.3.2	Slender body equations	7
2.3.3	Forces	7
2.3.4	Velocities	7
3	Serial Implementation	9
3.1	Time stepping	9
3.2	Quadrature	9
3.2.1	Numeric Integration	9
3.2.2	Analytic Integration	9
3.3	Linear system	9
3.4	Algorithm	9
4	GPU Programming	11
4.1	Overview	11
4.2	CUDA	14
5	Parallel Implementation	19
5.1	Development environment	19
5.2	Kernels	21
5.3	Optimizations	25
5.3.1	Numeric vs. Analytic Integration	26

5.3.2	Shared Memory	26
5.3.3	Thread Block Dimension	27
5.4	OpenMP	28
6	Benchmarks	31
6.1	Methodology	31
6.1.1	Hardware	31
6.1.2	Benchmark scheme	32
6.2	Optimizations	33
6.2.1	Numeric vs. Analytic Integration	34
6.2.2	Shared Memory	36
6.2.3	Thread Block Dimension	36
6.3	Linear Solvers	38
6.4	Comparing CPU and GPU performance	39
7	Experiments	43
7.1	Fiber Ring	43
7.2	Density effect on GMRES iterations	45
7.3	Spherical simulation	45
7.4	Density effect on spherical simulation	47
7.5	Number of fibers effect on spherical simulation	47
8	Conclusion	49
Appendices		49

List of listings

4.2.1 Pseudocode for CUDA vector addition	15
4.2.2 Pseudocode for CUDA matrix addition, illustrating 2D thread blocks	17
5.2.1 Pseudocode for parallel algorithm on the host.	21
5.2.2 Pseudocode for the assemble system step with a 1D thread block. .	22
5.2.3 Pseudocode for the updating velocities simulation step.	24
5.2.4 Pseudocode for the updating fibers simulation step.	25
6.1.1 Pseudocode for benchmark scheme.	33

Chapter 1

Introduction

Chapter 2

Theoretical Foundation

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

In this chapter we will present the theoretical foundation of the physics the simulations are based on. This is required to be able to understand the numerical method used throughout the rest of the thesis. However, the method is only presented in a reduced summary for more details please refer to [1].

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

2.1 Stokes flow and Stokeslet

The aim of the presented simulation is to model the behaviour of objects immersed in a fluid. We restrict ourselves to rigid bodies, which are slowly sedimenting in a viscous fluid due to gravity. The velocity of these rigid objects is only dependent on their orientation and the resulting drag. The interaction between multiple objects is only modeled through the surrounding fluid as collision are not modelled.

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \nabla \cdot \mathbf{u} = 0 \quad (2.1)$$

The Navier-Stokes equations describe the motion of fluids and allow to model the flow around an object. Using and solving these equations is however quite challenging due to their time dependence and non-linearity. Fortunately we can

make a number of simplification due to the restriction we placed on our model. Our model is characterized by three properties.

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

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

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

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

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

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

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

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

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

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

2.2. BOUNDARY INTEGRAL FORMULATION

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

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

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

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

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

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

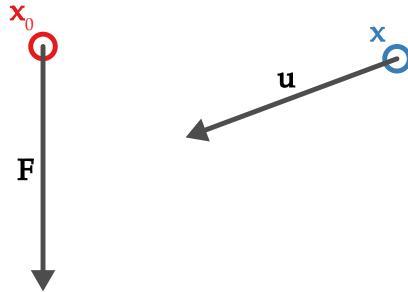


Figure 2.1: Singular point force embedded in stokes flow.

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

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

2.2 Boundary integral formulation

Using the Stokeslet from equation (2.6) we can now model the influence of an immersed object on the Stokes flow. For a given point y as well as the the force

distribution f on the surface of the object we can now calculate the resulting velocity at any point in the flow. The velocity u at the observation point x is then given by

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

which is illustrated in Figure 2.2.

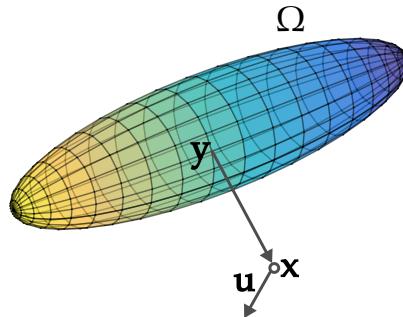


Figure 2.2: Immersed object in Stokes flow.

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

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

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

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

Furthermore we require that the integrated force distribution over the body as well as the torque have to be equal to the externally applied force. In our case this is simply gravity and an external torque of 0. For a single body the

2.3. SLENDER FIBERS

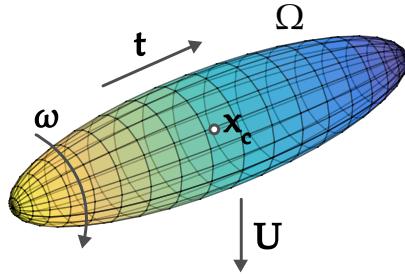


Figure 2.3: Immersed object in Stokes flow.

force \mathbf{F}_{body} and torque \mathbf{T}_{body} are thus defined as

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

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

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

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

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

2.3 Slender fibers

2.3.1 Nondimensionalization

2.3.2 Slender body equations

2.3.3 Forces

2.3.4 Velocities

Chapter 3

Serial Implementation

In the last chapter we presented the theoretical foundation of the physics and math involved in simulating rigid fibers. It showed how based on the Stokes Equation a framework can be developed to efficiently model rigid fibers.

Using this background we will now introduce the approach used for the numeric simulation. This is crucial to be able to validate the framework against real world experiments.

We will begin by presenting the employed time stepping approach. This is followed by a discussion about the different ways the integrals can be solved for the various quadrature points. The final section will illustrate the structure of the resulting final linear system and the briefly introduce the employed solvers.

3.1 Time stepping

3.2 Quadrature

3.2.1 Numeric Integration

3.2.2 Analytic Integration

3.3 Linear system

3.4 Algorithm

Chapter 4

GPU Programming

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

Based on the previous existing serial Fortran implementation this chapter will look at the algorithm in more detail and show, how it was adapted to take advantage of multi-core architectures. Modern GPUs in particular offer a massively parallel architecture and this is why the main focus of this thesis is the implementation of the simulation on a modern nVidia GPUs using CUDA. In addition to the main work of reimplementing the algorithm for CUDA and to have a better understanding of the achievable performance improvements the finished GPU code was also ported to multi-core CPUs using the OpenMP framework.

I will begin with a short introduction to general purpose computing on the GPU and explain briefly how CUDA works. I then move on to illustrate the practical implementation of the CUDA code. This is followed by a brief explanation of OpenMP and how the code was parallelized on the CPU. The chapter ends with the discussion of several potential optimization approaches to further improve the performance of the simulation.

4.1 Overview

In the beginning of Graphics Processing Units were highly specialized pieces of hardware developed to exclusively improve the performance of real-time 3D graphics. However, in recent years GPUs have started to be able to run arbitrary code instead of being limited to graphics related computations. This allows them to achieve impressive performance increases across a wide range of different

general purpose applications. The deciding factor is how well the problem can be parallelized to take advantage of the massively parallel architectures of GPUs. This has lead to potentially large performance advantages of GPUs over CPUs as illustrated in figure 4.1. It shows the year over year increase in the theoretical floating point operations per second (GFLOP/s). The FLOP/s number is calculated by combining information of the number of compute cores, core frequency and memory bandwidth for both the CPU and GPU models. This does not necessarily translate to direct real world performance increases but tries to show the potential GPUs have.

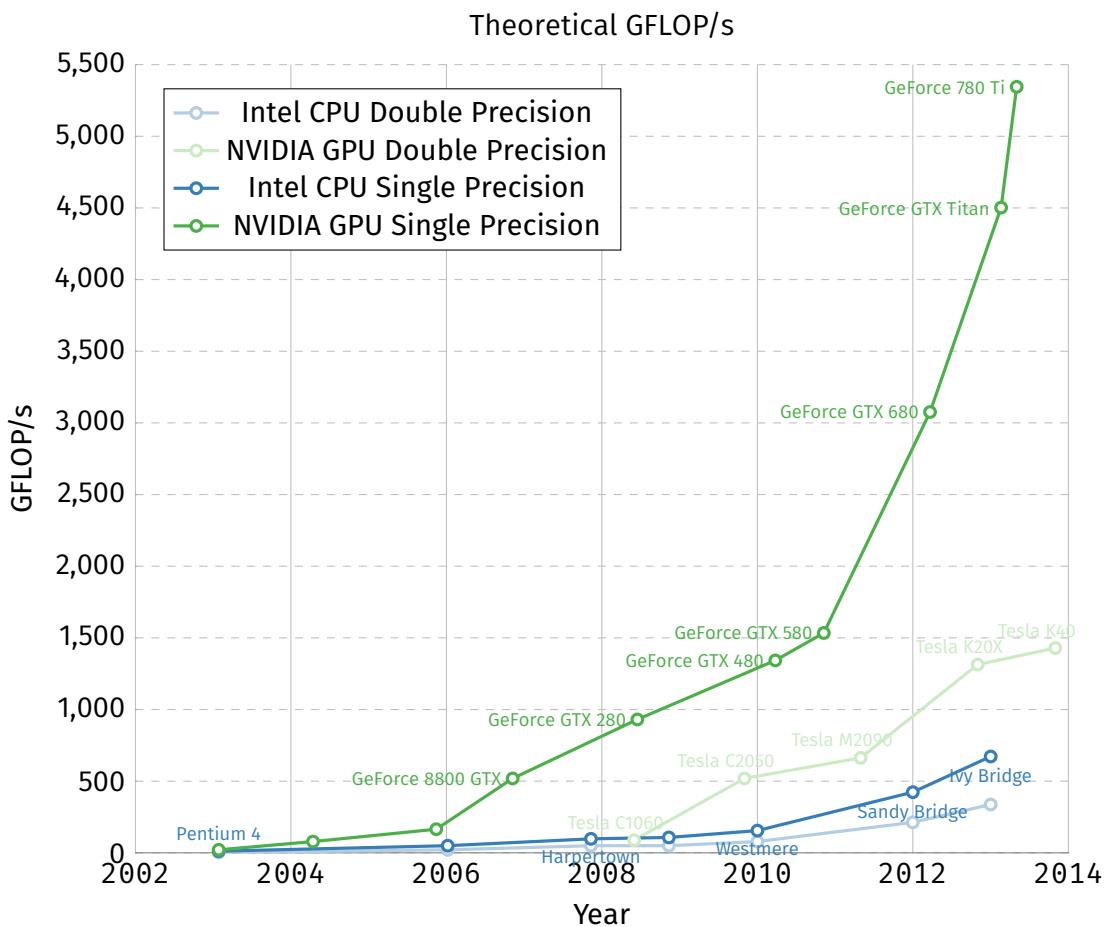


Figure 4.1: Increase in floating-point operations per second for CPUs versus GPUs over time.

The huge difference in performance mainly derives from the number of independent compute cores. Even though each individual core of a traditional CPU is

4.1. OVERVIEW

very fast they usually only have four, eight or maybe sixteen cores. In contrast to this GPUs can have several hundreds of independent compute cores. Each core can simultaneously perform calculations and thus provides the opportunity to yield big performance improvements for high-throughput type computations. This fact also introduced general purpose computing on GPUs to the world of supercomputers. Over time, a growing number of them started supplementing their compute power with GPUs and some even rely exclusively on GPUs for their computations.

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

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 be able 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 features in their new GPUs.

For this thesis I chose to go with Nvidia's CUDA framework mainly because of the available hardware both at the workstation computers as well as 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.

4.2 CUDA

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

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

The basic building blocks of the CUDA Programming Model from a development perspective are kernels. CUDA kernels are the equivalent of normal C functions. However the major difference is that instead of being executed just once, kernels are executed in parallel by N different threads. These CUDA threads are distributed and run across the available compute cores of the GPU. To illustrate how a very basic kernel call looks, figure 4.2.1 shows a code sample for a very simple vector addition.

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

Thread hierarchy In order to efficiently distribute the different threads across the compute cores of the GPU, CUDA defines a thread hierarchy. As discussed previously a GPU consists of many independent compute cores. On Nvidia GPUs these cores are referred to as Streaming Multiprocessors (SMs). During execution of the application each SM is tasked with running a distinct set of threads. In CUDA these

4.2. CUDA

GPU Computing Applications						
Libraries and Middleware						
CUFFT CUBLAS CURAND CUSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-Enabled NVIDIA GPUs						
Kepler Architecture (compute capabilities 3.x)	GeForce 600 Series	Quadro Kepler Series	Tesla K20 Tesla K10			
Fermi Architecture (compute capabilities 2.x)	GeForce 500 Series GeForce 400 Series	Quadro Fermi Series	Tesla 20 Series			
Tesla Architecture (compute capabilities 1.x)	GeForce 200 Series GeForce 9 Series GeForce 8 Series	Quadro FX Series Quadro Plex Series Quadro NVS Series	Tesla 10 Series			
	Entertainment	Professional Graphics	High Performance Computing			

Figure 4.2: Overview of the CUDA platform.

```

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 4.2.1: Pseudocode for CUDA vector addition

sets of threads are called thread blocks. Each thread block is then distributed to all the available SMs, which allows for automatic scalability depending on the number of SMs available as illustrated in figure 4.3.

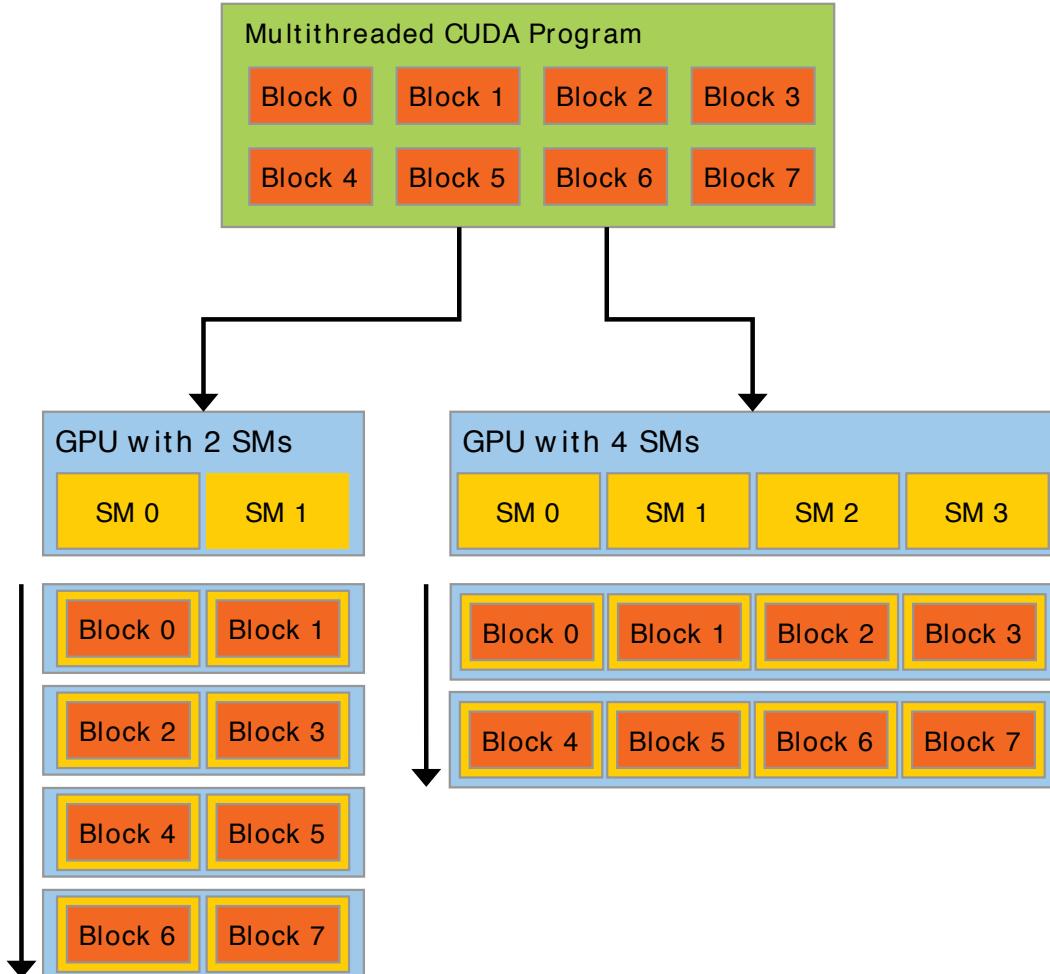


Figure 4.3: Automatic scaling of blocks across an arbitrary number of Streaming Multiprocessors.

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

In order to make programming and modeling of real world problems easier

4.2. CUDA

CUDA blocks can be addressed using either a one-dimensional, two-dimensional, or three-dimensional thread index. For example in the case of a matrix calculation it is more natural to think about parallelizing each element given by the row and column index instead of a single one-dimensional index. This is illustrated in the code simple in listing 4.2.2

```
1 // Kernel definition
2 __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
3 {
4     int i = threadIdx.x;
5     int j = threadIdx.y;
6     C[i][j] = A[i][j] + B[i][j];
7 }
8
9 int main()
10 {
11     ...
12     // Kernel invocation with one block of N * N * 1 threads
13     int numBlocks = 1;
14     dim3 threadsPerBlock(N, N);
15     MatAdd<<numBlocks, threadsPerBlock>>(A,B,C);
16     ...
17 }
```

Listing 4.2.2: Pseudocode for CUDA matrix addition, illustrating 2D thread blocks

Finally because the resources of each Streaming Multiprocessor are limited there exists a upper bound of how many threads a block can contain. Currently this maximum number of threads is 1024. This means that the maximum size of matrices possible to be added in the code sample in listing 4.2.2 is 32×32 . To solve this problem CUDA introduces another layer above blocks called a grid. A grid organizes thread blocks again into either one, two, or three dimensions. The number of thread blocks in a grid is unlimited and thus solely dependent on the size of the workload. Listing 4.4 shows an example configuration of a 2D grid with 2D blocks.

Memory hierarchy In addition to the Thread hierarchy...

- 4 Layers, Global, Local, Private
- Global shared across all SM
- Local shared across thread block

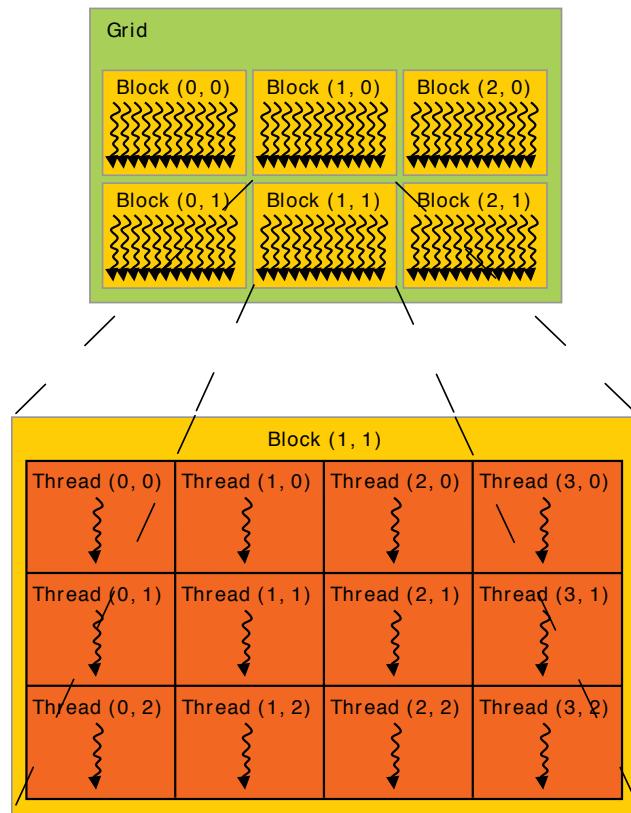


Figure 4.4: Overview of the CUDA platform.

- Private per thread
- Latency VERY different between layers
- Avoid global memory access
- Or hide with compute heavy, as is the case with assemble system step (ala will be used later)

Chapter 5

Parallel Implementation

5.1 Development environment

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

The development was done exclusively on a Linux workstation running Ubuntu as this will also be the exact same runtime environment used in the later experimental usage of the resulting application. The build system for compiling and linking the final application was CMake. It was chosen because it is widely used open-source and cross-platform build system, which allowing for easy integration of the various required libraries in a well documented and straightforward manner.

Under the hood the build system uses Nvidia's CUDA platform tools to compile the code. For this Nvidia includes *nvcc*, an LLVM-based CUDA compiler capable of compiling C/C++ code together with the CUDA specific extensions. In addition to the CUDA platform, the application also requires support libraries for the different linear solvers. The two main libraries are *MAGMA* for the direct solver and *ViennaCL* for the iterative solvers. Both will be introduced briefly now.

MAGMA / CuBLAS / OpenBLAS The MAGMA project contributes the implementation for the direct solver used in this thesis. This dense linear algebra library provides features similar to standard LAPACK functions but for multicore architectures. It also has features to support hybrid algorithms to run code across multiple

GPUs or CPUs at the same time, however these features were not explored in this thesis. Instead the focus was on a high performant single GPU implementation of a direct linear system solver.

MAGMA provides the interfaces for various high-level languages, however the underlying math functions utilize the platform specific implementations of the BLAS levels. For CUDA this is implemented directly by Nvidia in the form of the CuBLAS libraries. Additionally, MAGMA tries to be as fast as possible which sometimes means integrating CPU based algorithm where it has performance advantages. Thus a CPU based BLAS implementation is also needed. For this the OpenBLAS library was chosen which is the most up-to-date and high performant library available outside the very expensive Intel MKL library. OpenBLAS takes full advantage of multicore systems using pthreads and is also used for the comparison of Fortran CPU implementation against the CUDA GPU implementation

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

ViennaCLs 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 both the BiCGStab as well as the GMRES iterative solvers were used and tested.

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

5.2. KERNELS

aggregating timings for each simulation step as well as the total time.

- Unified interface for OpenMP and CUDA -> OpenMP only introduced later...
-> Is that really important any way?

5.2 Kernels

The overall parallel algorithm is very similar to the serial version, however, each simulation step is separated into different kernels. Each kernel is invoked in a serial manner, this means CUDA guarantees that all data modified in a kernel is available before the next kernel is executed. These kernels are then distributed across the GPU. All calculations are done using single precision floating point numbers, as Nvidia limits high performance double precision computation to their server class GPUs. The CUDA pseudocode for the algorithm is illustrated in listing 5.2.1. It begins by parsing 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 timesteps and for each step the four main algorithm substeps – 1. *Assemble System*, 2. *Solve System*, 3. *Update Velocities* and 4. *Update Fibers* – are run on the GPU.

```
1 int main()
2 {
3     // Parsing algorithm parameters and initial fiber positions
4     readParameters();
5     readFiberConfiguration();
6     allocateGPUMemory();
7     ...
8
9     for (int step = 0; step < max_timestep; step++)
10    {
11        AssembleSystem<<numBlocks, threadsPerBlock>>(...);
12        SolveSystem<<numBlocks, threadsPerBlock>>(...);
13        UpdateVelocities<<numBlocks, threadsPerBlock>>(...);
14        UpdateFibers<<numBlocks, threadsPerBlock>>(...);
15    }
16    ...
17 }
```

Listing 5.2.1: 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 timesteps as well as the number of force expansion terms and number of 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 substeps are now discussed in more detail. The purpose of each kernel as well as the required input and outputs.

```

1 __global__ void AssembleSystem1D(
2     in float *positions,
3     in float *orientations,
4     out float *a_matrix,
5     out float *b_vector)
6 {
7     const int i = blockIdx.x * blockDim.x + threadIdx.x;
8
9     if (i >= NUMBER_OF_FIBERS) return;
10
11    for (int j = 0; j < NUMBER_OF_FIBERS ++j)
12    {
13        for (int force_index_j = 0;
14             force_index_j < NUMBER_OF_TERMS_IN_FORCE_EXPANSION;
15             ++force_index_j)
16        {
17            computeInnerIntegral(...);
18
19            for (int force_index_i = 0;
20                 force_index_i < NUMBER_OF_TERMS_IN_FORCE_EXPANSION;
21                 ++force_index_i)
22            {
23                // Only 1D thread block
24                // Each thread updates unique memory locations, thus
25                // no need for atomics
26                setMatrix(...)
27                setVector(...)
28            }
29        }
30    }
31 }
```

Listing 5.2.2: Pseudocode for the assemble system step with a 1D thread block.

Assemble System The *Assemble System* kernel is the most important step of the algorithm. Its goal is to build the matrix and vector in memory for the linear system of equations in the form of $A * x = b$. Listing 5.2.2 shows the pseudocode for the one-dimensional implementation of the assemble system step. This means the

5.2. KERNELS

code is parallelized for each fiber and each thread calculates the contributions to this fiber from all other fibers. Looking at the matrix each thread is thus responsible for $3 * M$ rows of the matrix.

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

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

Update Velocities After solving the linear system the solution coefficients are used to update the velocities of the fibers. The *Update Velocities* kernel accumulates the exerted forces for all fibers and updates both the translational as well as the rotational velocities simultaneously. In this particular instance the 2D thread block version of the kernel is illustrated in listing 5.2.3. This means each individual kernel invocation is responsible for a single pair of fiber interaction. Under the normal assumption that kernel invocations are not allowed to write to the same memory location since this would result in undefined behaviour and incorrect results for the velocities.

Fortunately CUDA provides a mechanism to circumvent this issue. By using so-called atomic function CUDA streamlines and serializes the memory access. Thus the atomicAdd function accumulates different velocity contributions to a fiber from all the other fibers. This ensures that each update to the memory location is handled in a serial manner, guaranteeing the correct value in memory for each. Of course this implies a potential performance degradation, however, newer GPUs with new CUDA version have been very well optimized to only have a minimal negligible impact. Benchmarking for the fibers simulation shows that

```

1 __global__ void UpdateVelocities2D(...)
2 {
3     const int i = blockIdx.x * blockDim.x + threadIdx.x;
4     const int j = blockIdx.y * blockDim.y + threadIdx.y;
5
6     if (i >= NUMBER_OF_FIBERS) return;
7     if (j >= NUMBER_OF_FIBERS) return;
8     if (i==j) return;
9
10    for (int quadrature_index_i = 0;
11         quadrature_index_i < TOTAL_NUMBER_OF_QUADRATURE_POINTS;
12         ++quadrature_index_i)
13    {
14        for (int quadrature_index_j = 0;
15             quadrature_index_j < TOTAL_NUMBER_OF_QUADRATURE_POINTS;
16             ++quadrature_index_j)
17        {
18            force = computeForce(coefficents, ...)
19            computeDeltaVelocities(force)
20        }
21    }
22
23    // 2D thread block
24    // Each thread responsible for an interaction pair, thus
25    // result is written to the same memory location
26    // Using atomics to avoid conflicts
27    atomicAdd(&(translational_velocities[i].x),
28              delta_translational_velocity.x);
29    atomicAdd(&(translational_velocities[i].y),
30              delta_translational_velocity.y);
31    atomicAdd(&(translational_velocities[i].z),
32              delta_translational_velocity.z);
33
34    atomicAdd(&(rotational_velocities[i].x),
35              delta_rotational_velocity.x);
36    atomicAdd(&(rotational_velocities[i].y),
37              delta_rotational_velocity.y);
38    atomicAdd(&(rotational_velocities[i].z),
39              delta_rotational_velocity.z);
40 }
```

Listing 5.2.3: Pseudocode for the updating velocities simulation step.

5.3. OPTIMIZATIONS

using a 2D thread block and the associated performance increases far outweigh the potential performance hit of using atomics.

```
1 __global__ void UpdateFibers()
2 {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4
5     if (i >= NUMBER_OF_FIBERS) return;
6
7     next_positions[i] = 4/3 * current_positions[i]
8         - 1/3 * previous_positions[i]
9         + 2/3 * Timestep
10        * (2 * current_translational_velocities[i]
11            - previous_translational_velocities[i]))
12
13    next_orientations[i] = 4/3 * current_orientations[i]
14        - 1/3 * previous_orientations[i]
15        + 2/3 * Timestep
16        * (2 * current_rotational_velocities[i]
17            - current_rotational_velocities[i]))
18
19    normalize(next_orientations)
20 }
```

Listing 5.2.4: Pseudocode for the updating fibers simulation step.

Update Fibers The final simulation step takes care of advancing the position and orientation of the fibers in time. The pseudocode in listing 5.2.4 implements the second-order multi-step method introduced in section 3.1. As will be seen later in the results in chapter 6 this required time for this kernel is minuscule compared to the other steps. The kernel only scales linearly and in addition has a perfectly aligned memory access resulting in close to optimal usage of the GPU hardware.

5.3 Optimizations

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

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

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

Many optimizations performed during this process are applicable to both the CPU and GPU and show performance improvements for both. However, some optimizations or algorithm variations have either a different effect or are uniquely suited and related to the GPU hardware. For this thesis we will look into three general different optimizations in more detail. The performance results for each will then later be discussed in chapter 6.

5.3.1 Numeric vs. Analytic Integration

The first optimization was already part of the original serial implementation as described in section 3.2. In the original paper [] the authors observed that the analytical integration of the inner integral yielded a performance increase compared to the numerical integration. On paper an analytical integration should also not only be preferred because of being faster but more importantly also being more accurate.

However, for numerical precision reasons the actual implementation of the analytical integration can't achieve this theoretical level of accuracy. The implementation is dependent on a design variable which uses a slightly different code to calculate the integral for very close fibers in order to avoid numerical instabilities. The computation of the inner integral lies at the heart of the algorithm and is a very performance critical section of the implementation. Exploring the performance implications of both approaches on the GPU is thus of great interest, and especially how it compares against the same algorithm running on the CPU.

5.3.2 Shared Memory

As the described in section 4.2 CUDA code is subject to a highly specialized memory hierarchy. Whereas traditional CPUs only have small caches and a large main memory pool, CUDA introduces the concept of a shared local memory space. The access time to this local memory is orders of magnitudes faster compared to

5.3. OPTIMIZATIONS

accessing the global GPU memory. Additionally, local memory can be shared among a CUDA thread block and potentially save time by avoiding to constantly access the slow global memory.

In order to test this, shared memory was implemented and tested with the *Assemble System* step, the most performance critical kernel written for the fiber simulation. To understand the idea, imagine the 2D thread block implementation of the kernel. Each thread block is responsible for many pairs of fiber interactions, e.g. fibers [1, 8] each interacting with fibers [9, 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, however, is not always easy to tell, as various factors can influence the result. If the loaded data is small enough, the various memory caches can hide the increased costs of accessing global memory. Also while the first threads in a thread block load the data from global to shared memory, all other threads in this thread block have to wait for their data to become available. This has a particular large impact when the overall kernel has to execute many calculations and the required time for computations effectively hides the memory loading time. CUDA is then able to seamlessly and very efficiently switch to the next thread block and continue executing it. In this case the performance penalty only occurs at the very beginning and end, and has only a negligible impact on the overall performance.

However, in general efficient exploitation of shared memory can be a huge advantage for parallel GPU implementations. Especially when comparing the performance against CPUs, as they don't have an equivalent fast and comparatively large memory space. It is therefore an interesting optimization to explore and benchmark.

5.3.3 Thread Block Dimension

There are many different factors that determine the performance of a particular GPU algorithm. This is especially true with regards to optimally taking advantage of the specific underlying GPU architecture, which change even between different models of graphics cards. How to best utilize the hardware depends on specific

memory access patterns, avoiding too much register usage and choosing optimal settings for the thread block size. Each graphics cards can have a different number of streaming multiprocessors with only limited resources and taking advantage of this can result in performance increases.

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

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

For a two-dimensional thread block each kernel invocation is responsible for a pair of fibers interacting. While this decreases the necessary resources, it also required atomics which can potential slow down the execution. Three-dimensional thread blocks are the maximum allowed dimensions for a CUDA thread block. They are a further extension of the two-dimensional thread block, as now each kernel invocation is responsible not for the complete interaction but only the interaction resulting from a specific point from 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.

How exactly each decision for the thread-block dimension affects performance is not clear. Only trial-and-error benchmarking combined with metrics from CUDA can find the optimal setting for the specific algorithm.

5.4 OpenMP

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

5.4. OPENMP

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

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

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

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

Chapter 6

Benchmarks

The last chapter introduced the parallel implementation of the numerical simulation. It presented the concept of general purpose computing on modern GPUs. This was followed with a practical overview of the implementation of the algorithm using Nvidia's CUDA framework. Additionally, possible optimizations to take advantage of the unique properties of the GPU architecture were outlined.

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

6.1 Methodology

The methodology used for the benchmark suite is the same for all presented benchmarks. This ensures comparable results and fairest comparison possible.

6.1.1 Hardware

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

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.xxxx

Table 6.1: Benchmark system hardware specification

6.1.2 Benchmark scheme

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

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

Using this semi-random generation the rigid fiber simulation is run for exactly 10 timesteps. To avoid remaining outliers in the configuration causing potentially large variation in the timings the first timestep is excluded. This is especially true for iterative solvers which are highly dependent on the configuration. The first timestep is thus used as a simple warmup step for the simulation. So the final average time for each run is taken from the last 9 timesteps.

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

6.2. OPTIMIZATIONS

than a specified threshold, the number of iterations is doubled and run again.

```
1  for(int N = 100; N <= 2000; N += 100)
2  {
3      int iterations = 4;
4      while (iterations <= MAX_ITER)
5      {
6          for(int i = 0; i < iterations; ++i)
7          {
8              generateRandomInitialFiberConfiguration();
9              run(10); // execute 10 timesteps
10             collectTimings();
11         }
12
13         rse = calculateRelativeStandardError();
14
15         if (rse <= 0.2)
16         {
17             break;
18         }
19
20         iterations *= 2
21     }
22
23     reportTimings();
24 }
```

Listing 6.1.1: Pseudocode for benchmark scheme.

The benchmark suite uses a minimum of 4 iterations to obtain the initial timings in case the relative standard error is not below a threshold of 20% an additional 4 iterations are run to bring the total to 8. This process repeats until the threshold is not exceeded and more reliable benchmark timings have been obtained. The algorithm for collecting the benchmark results is illustrated using pseudocode in listing 6.1.1.

6.2 Optimizations

We now look at the performance results for the different optimizations previously outlined in section ???. Where applicable the results will be compared between the OpenMP and 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. It can be solved either numerically or analytically. Figure 6.1 illustrates the performance timings for the *Assemble System* step of the parallel OpenMP version. Inline with the observations made by the authors of the original serial implementation the parallel version of the analytic integration is always faster than the numeric integration.

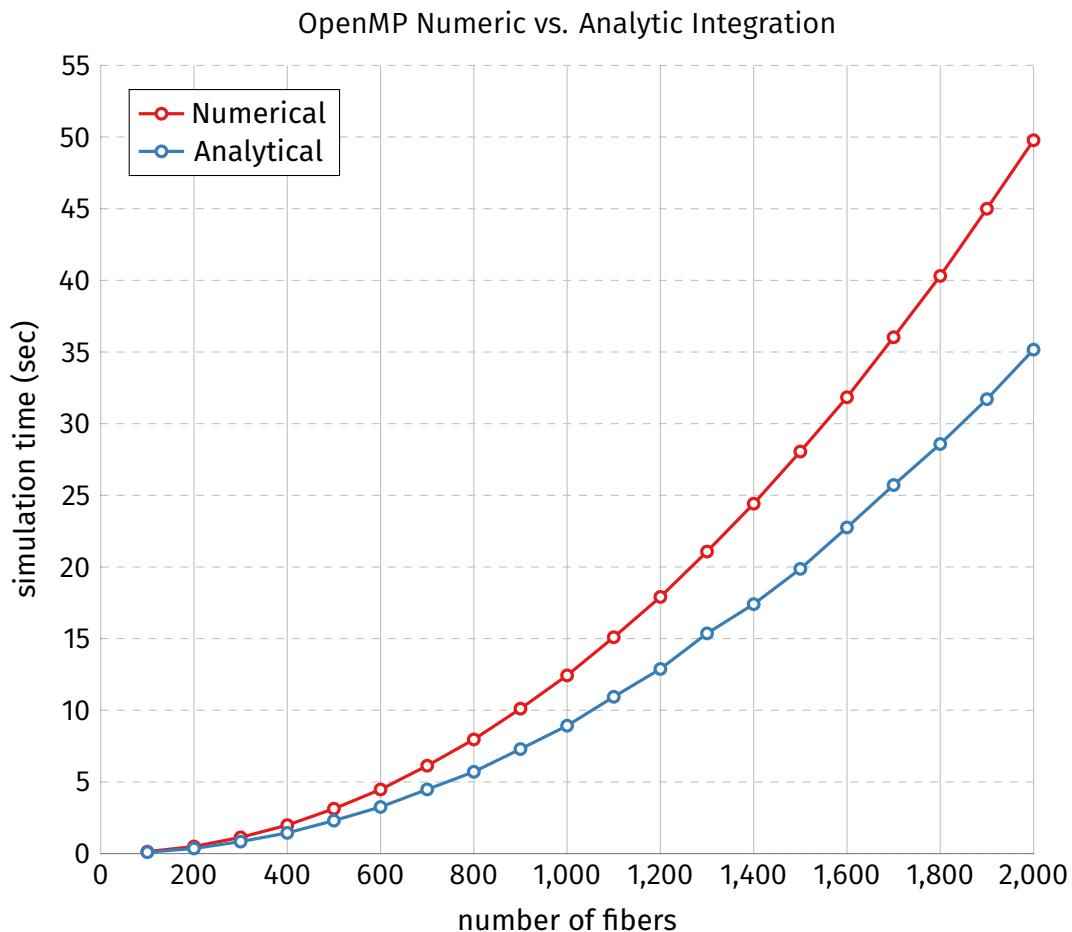


Figure 6.1: Benchmark of assemble system step for integration of inner integral.

However, the picture is more interesting when we look at the same graph for the CUDA implementation in figure 6.2. Here the results are reversed. The numeric integration outperforms the analytical integration by a large margin increasing with the number of fibers.

6.2. OPTIMIZATIONS

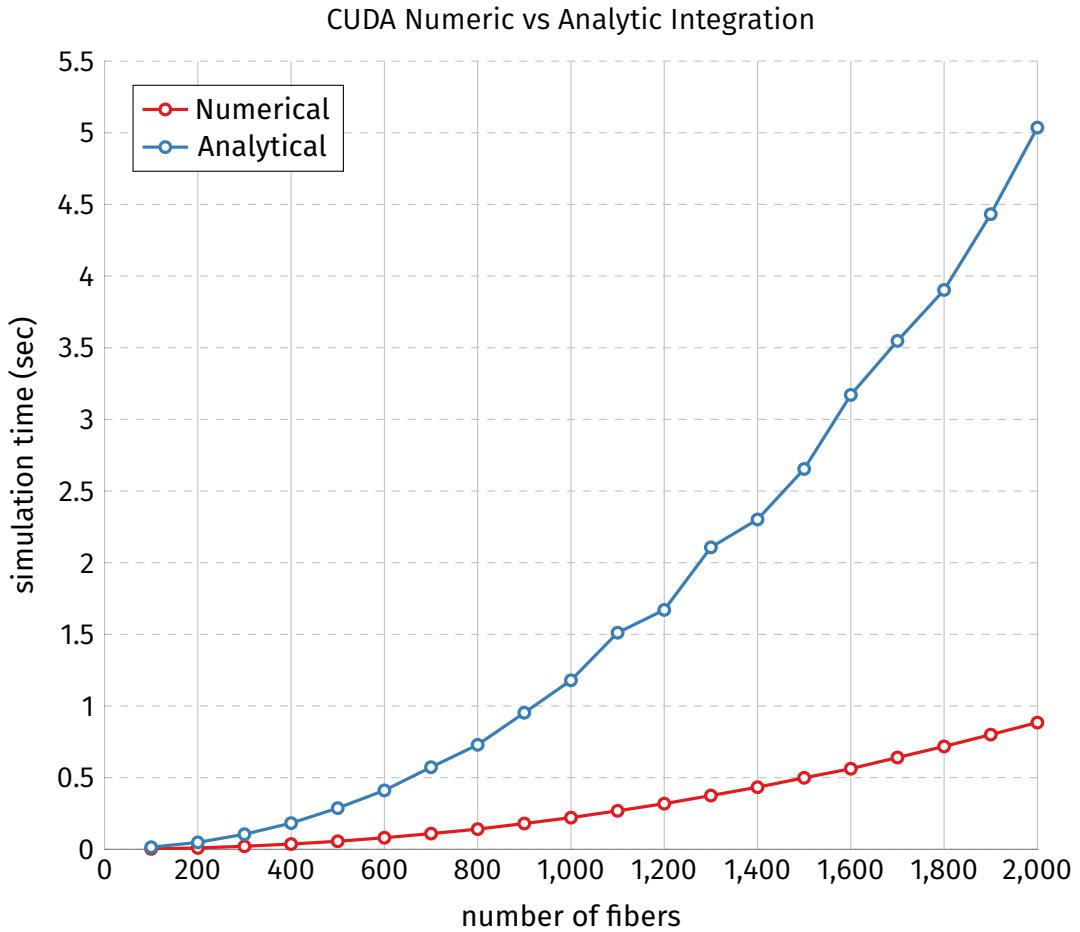


Figure 6.2: Benchmark of assemble system step for integration of inner integral.

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

To confirm that Branch Divergence is the reason for the slow down of the

analytic integration on the GPU we look at the metrics of the CUDA profiler *nvprof*. The metric *Warp Execution Efficiency* shows the ratio of the average active threads per warp to the maximum number of threads per warp. The metrics for both the numerical and analytical integration of the Spherical fiber setup used in section ?? can be seen in table 6.2.

Algorithm	warp_execution_efficiency
Numerical	99.01%
Analytical	53.79%

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

On one hand the numerical integration is almost 100% efficient, meaning all warps execute in complete lockstep. The analytical integration on the other hand is only 50% efficient, meaning that most of the time only half of the threads actually perform work while the other half is just waiting. This results in the observed performance difference. Closer inspection of the source code reveals that the design constant and the resulting branching as described in section ?? is responsible for the divergence. The constant determines if a slightly different code should be executed in case fibers are very close together. Unfortunately, in order to ensure numeric stability this workaround is unavoidable.

6.2.2 Shared Memory

- No effect of the performance - Assemble System is Compute Bound not Memory Bound. The transfer times are dwarfed by the compute time

6.2.3 Thread Block Dimension

The next optimization looked at was the Thread Block Dimension on the GPU. Choosing the best option is a trade-off between the resources used and the overhead caused by an increased amount of memory writes to the same location and thus the need for potential slow atomics.

The results in figure 6.3 indicate that the best option for this particular GPU is a two-dimensional thread block. Using a three-dimensional thread block is always slower and has a worse scaling factor. Using the *Atomic Transactions* from *nvprof* shows the total number of atomic transactions that had to be performed. As can

6.2. OPTIMIZATIONS

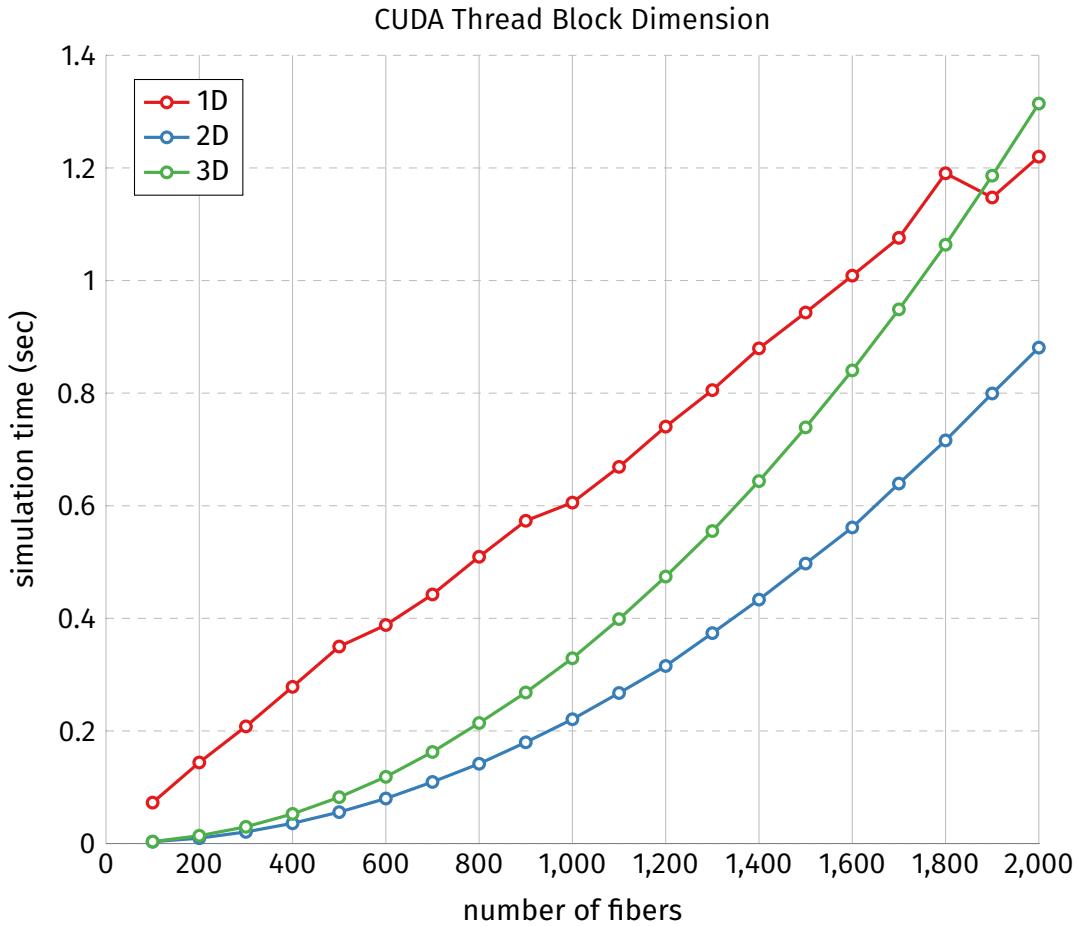


Figure 6.3: Benchmark of assemble system step for different thread block dimensions.

be seen in table 6.3, the required atomic transactions for the sphere example from section ?? are almost two times larger in the 3D case.

Algorithm	atomic_transactions
2D	1269325
3D	2350670

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

The one-dimensional approach is also slower than either two-dimensional or three-dimensional. However, it appears to scale linearly whereas the other two scale exponentially. It can already be observed that 1D becomes faster than 3D

for close to 2000 fibers. Unfortunately, the hardware of the workstation does not have enough memory to simulate more fibers.

6.3 Linear Solvers

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

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

Up until around 500 fibers GMRES is indeed faster than as reported by the original authors. However, after that the direct solver starts to perform much better, being close to two times faster. Additionally it can be seen that the iterative GMRES solver is fluctuating widely owing to the fact that the performance is highly dependent on the particular tested initial fiber configuration. As the direct solver always performs the same number of computations it doesn't suffer from this. Furthermore, the direct solver has the advantage to always calculate the exact result excluding numerical precision as compared to the close solution iterative solvers provide.

On the GPU side we used the direct solver provided by the MAGMA library. For the iterative solvers both GMRES as well as BiCGStab were compared as they can be easily exchanged and tested using the ViennaCL library. The benchmark results for the CUDA solvers are illustrated in figure 6.5.

The same fact that the direct solver is faster than the iterative solvers holds true for the GPU. However, compared to the CPU solvers the difference between the performance of the direct solver against the iterative solvers is even more

6.4. COMPARING CPU AND GPU PERFORMANCE

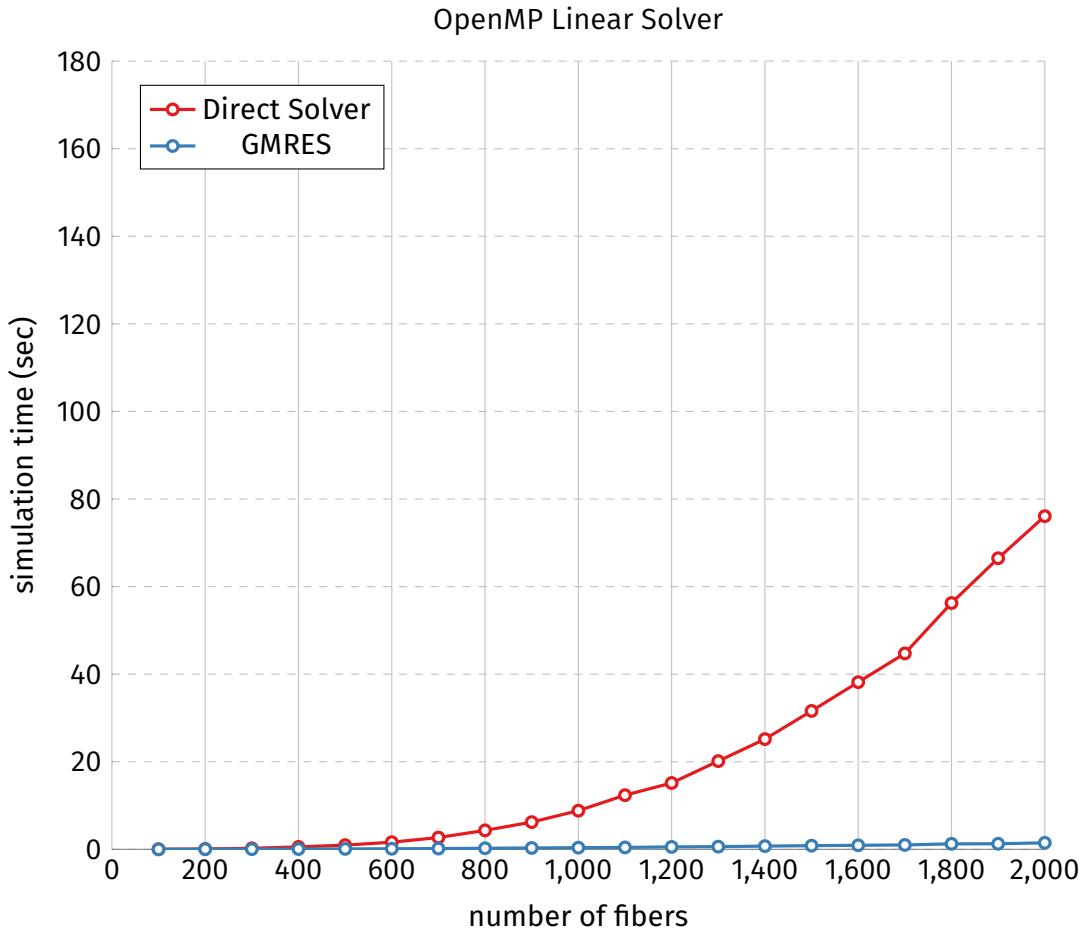


Figure 6.4: Benchmark of solve system step for different Fortran solvers.

pronounced. At close to 2000 fibers the direct solver is about three to four times faster than either GMRES or BiCGStab. Looking at the difference between the two iterative solvers BiCGStab is slightly faster than GMRES, whether this is an inherent property of the algorithm or simply a more performant implementation is not clear. Either way the direct solver is not only the clear winner in terms of performance but also provides exact solutions.

6.4 Comparing CPU and GPU performance

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. The underlying architectures

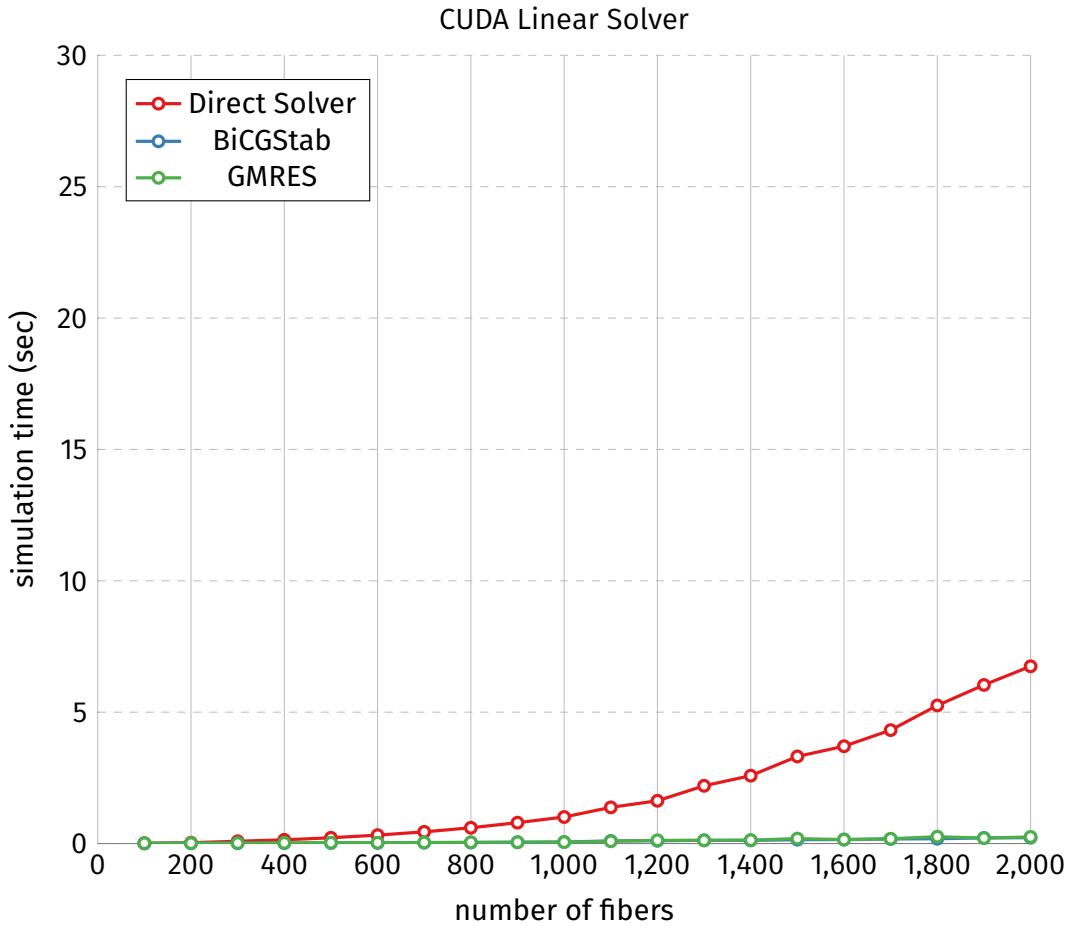


Figure 6.5: Benchmark of solve system step for different GPU linear solvers.

of the two approaches are completely different and thus not really comparable. Some try to extrapolate relative performance from the underlying FLOPs by taking processor count, frequency and memory bandwidth into account. However, due to of integrate hardware details this approach is also not applicable to all scenarios. Thus in the super computing community metrics like performance-per-dollar or even performance-per-watt have become the main focus.

Exploring this question in more detail is out of the scope of this thesis. In order to come as close as possible given these complexities and constraints, we used both a current CPU and GPU considered as a balanced system at the time of writing. Additionally, we implemented the parallel OpenMP version. It is directly based on the parallel CUDA version with the sole purpose to have as few

6.4. COMPARING CPU AND GPU PERFORMANCE

difference between the two implementations as possible. Furthermore, the final benchmark for each uses the fastest possible algorithm variant as determined by all previously performed benchmarks to compare the best variant for each architecture.

The results for average time required to take a single timestep is illustrated in figure 6.6. For OpenMP the algorithm uses the analytical integration of the inner integral. The linear system is solved by the direct solver provided by OpenBLAS. For CUDA, we used the numerical integration and the thread block dimension was chosen to be two-dimensional. The direct solver for MAGMA is used to solve the linear system.

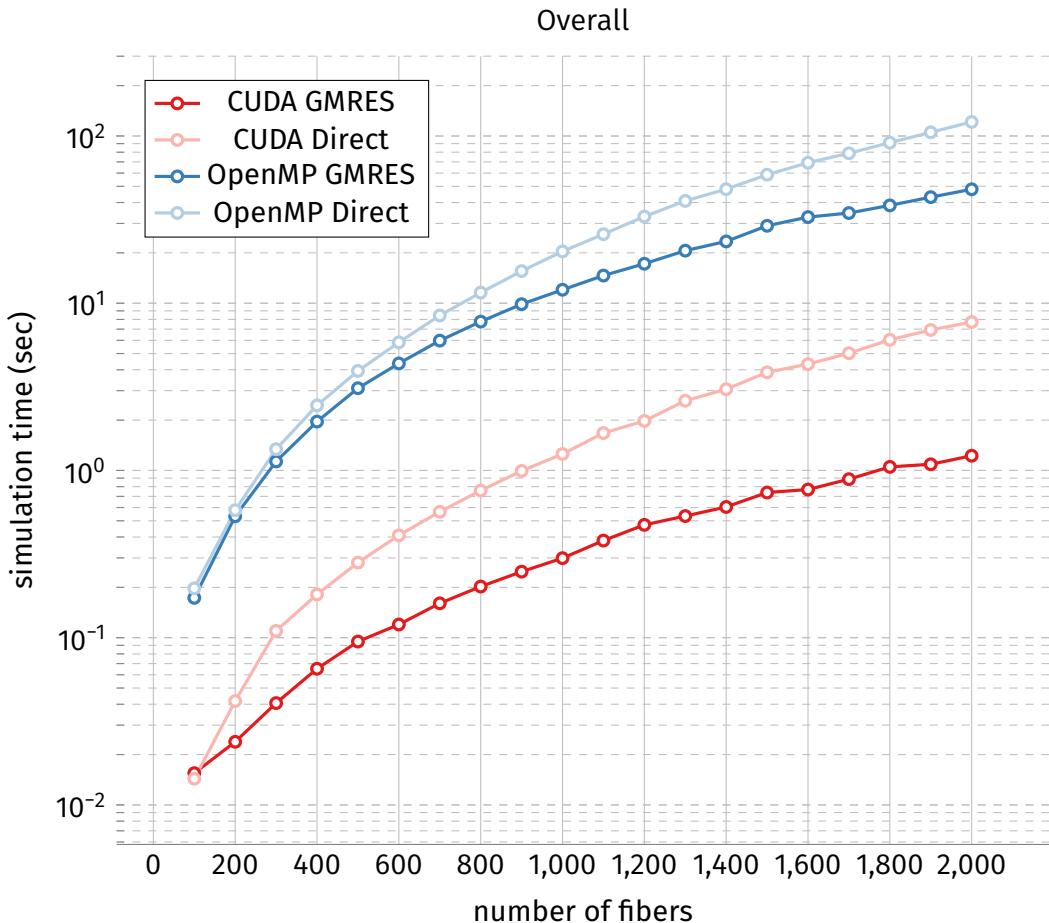


Figure 6.6: Benchmark of overall timestep for both OpenMP and CUDA.

The required simulation time on the GPU outperforms the CPU by a wide margin. The GPU version is faster for any number of fibers. CUDA maintains a relative

performance of $15\times$. The only advantage the CPU version has is the potentially larger memory as 4GB on the GPU limits the number of fibers to roughly 2000, so for more fibers OpenMP is the only option.

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

Table 6.4: Overall speedup factor for 2000 fibers.

We acknowledge that these numbers and performance increases is 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. No need to wait for computing time at a computing cluster but instead simple and fast iterations using a workstation. The observed performance increase of $15\times$ is a difference between a whole day of waiting for the simulation results and a quick 1.5 hour result during the day. The saved time also translates directly into the ability to simulate many more fibers than ever before. Before the largest simulation possible in a reasonable time frame where around 500 fibers, in contrast one single timestep for 2000 takes only about 10 seconds on the GPU. Therefor the implementation on the GPU and the optimizations of the simulation code is a great value to the research of rigid fiber simulations.

Chapter 7

Experiments

Before examining the different performance metrics and optimizations we will first look at a couple of examples to better illustrate the rigid fiber simulation. The first example shows the numerical precision of the final single precision simulation compared against both the original serial double precision and the expected physical results. The second example presents a real world example which illustrates an interesting physical phenomenon. Both were run with the fastest CUDA algorithm implemented in the thesis.

7.1 Fiber Ring

The first example was used to verify that the single precision GPU version is able to replicate the result of the original double precision code. For this we looked at a very simple problem where a small number of fibers are setup perfectly symmetrical position and orientation. All fibers are evenly distributed on a circle and aligned vertically with gravity. During the simulation the fibers begin to rotate from their vertical orientation towards the horizontal position. Afterwards they continue rotating back into the vertical position. This motion is referred to as tumbling orbits and as long as there are no disturbances or numerical precision issues it repeat forever.

This simple but very interesting problem has also been simulated in []. Additionally an even more simplified version with only two fibers was studied numerically and experimentally by []. This example is thus ideally suited to test and verify the numerical precision of GPU simulation. A visualization of the GPU simulation of 16 fibers evenly distributed around a circle with diameter 1.1 is shown in figure 7.1.

In order to analyse the periodic movement of the fibers we now look at the

CHAPTER 7. EXPERIMENTS

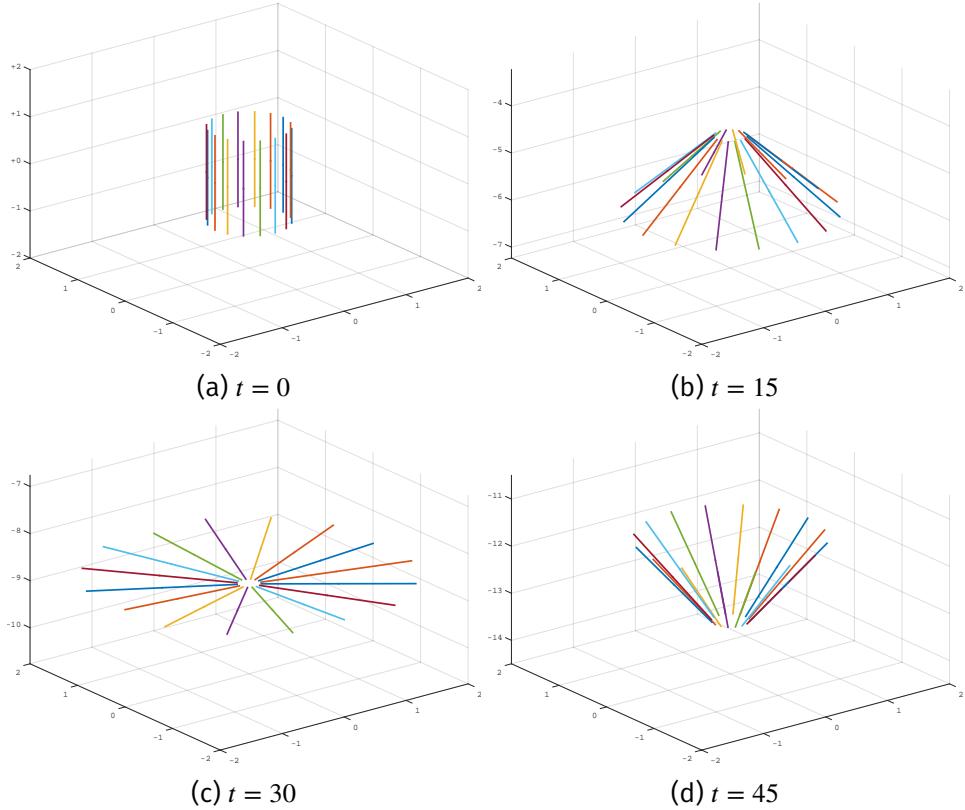


Figure 7.1: Visualization Ring Simulation

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

Figure 7.2 shows a graph of the sedimentation velocity of a single fiber overtime. As the same force acts on all fibers and they perform the same motion they all have the same sedimentation velocity. For this particular setup the maximum velocity is 3.8 and the minimum velocity is 2.2. The graphs clearly shows the periodical rotation the fiber perform. This result perfectly captures the expected result obtained from prior simulation and experiments.

7.2. DENSITY EFFECT ON GMRES ITERATIONS

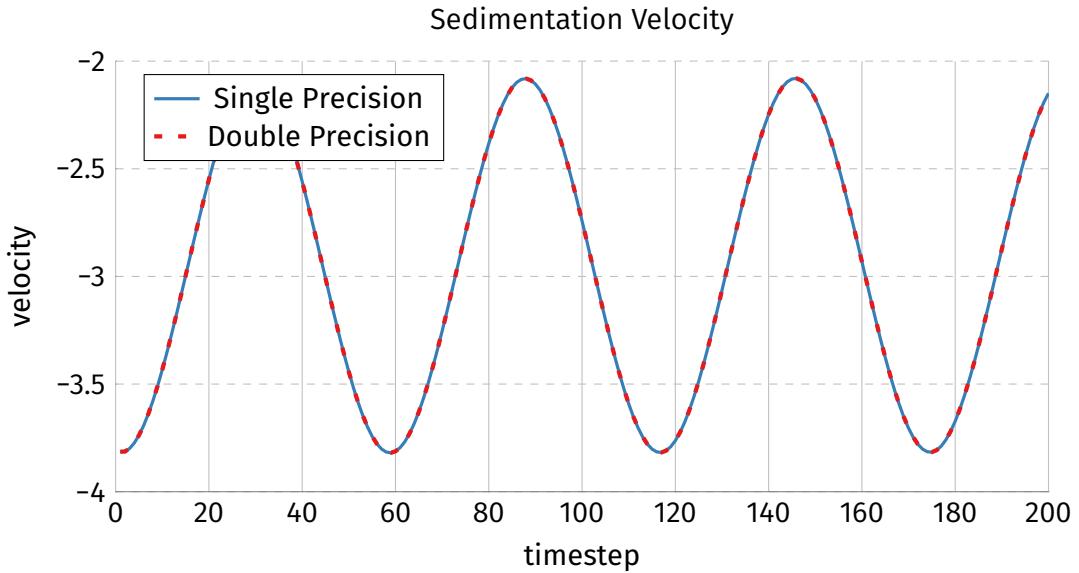


Figure 7.2: Comparison of sedimentation velocity for single- and double-precision simulation.

7.2 Density effect on GMRES iterations

7.3 Spherical simulation

The second example showcases a more chaotic system with large number of interacting fibers. The observed phenomena only occurs if enough fibers are simulated and is thus ideally suited for the GPU simulation as it reduces the required time per step. For this example 500 fibers are initially distributed inside a sphere with a radius of 3. Both their positions and orientations are randomly generated inside this sphere. An illustration of an example run can be seen in figure 7.3 and the corresponding configuration parameters in table 7.1.

This sphere have packed fibers is then allowed to sediment due to gravity. Beginning from this spherical shape the interacting fibers begin to form a ring similar to the result shown in the previous example in section ???. However due to the large number of fibers and random initial setup the tumbling orbits evident and the behaviour is more chaotic.

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

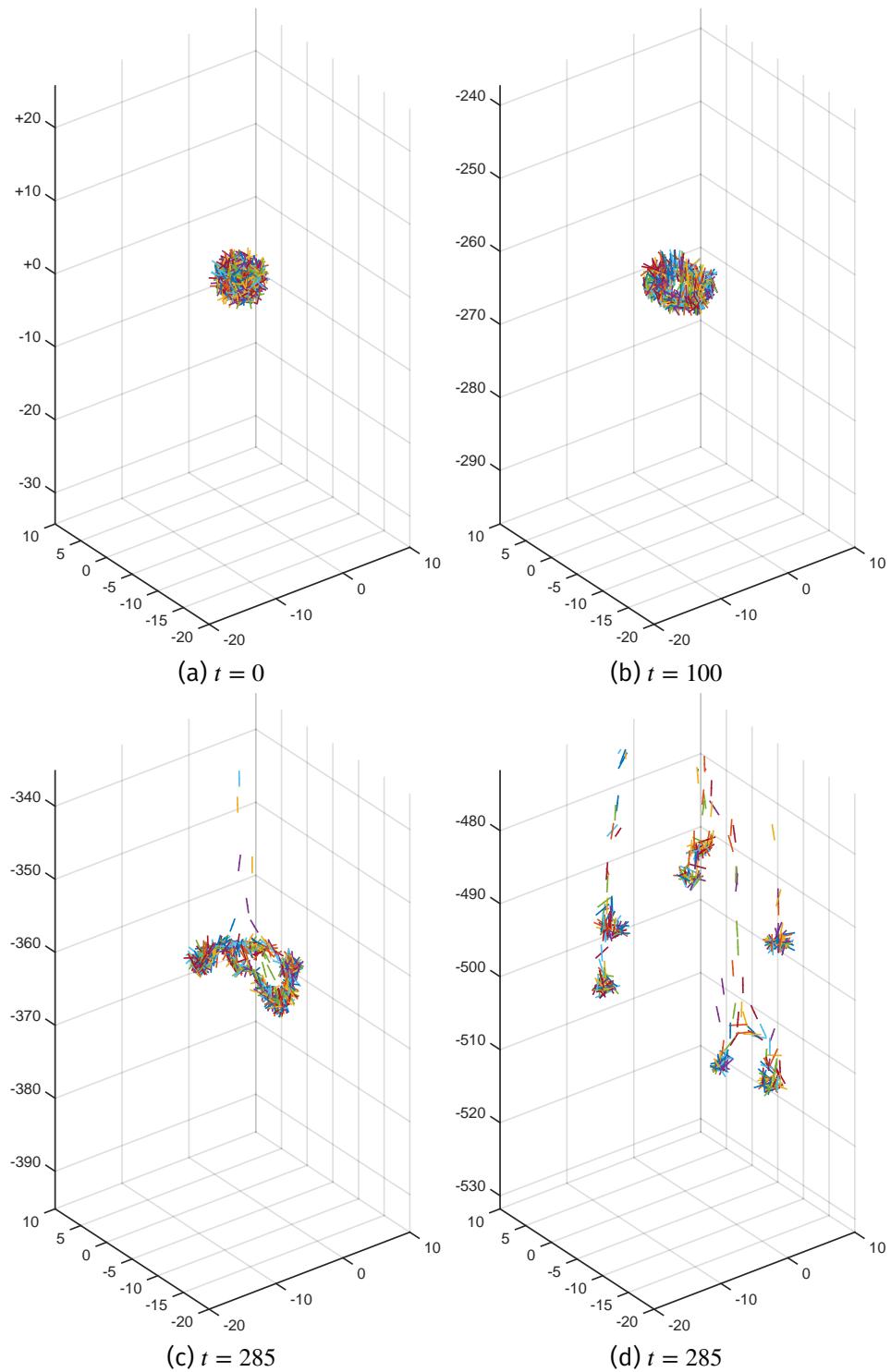


Figure 7.3: Visualization Sphere Simulation

7.4. DENSITY EFFECT ON SPHERICAL SIMULATION

Parameter	Value
Sphere Radius	3
Number of fibers	500
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

Table 7.1: Parameters for example sphere simulation.

For the presented example with the GPU simulation only takes 0.28 seconds. It is able to perform the simulated 300 timesteps in less than 2 minutes. The new GPU simulation can now be used to increase the number of fibers even further and explore the intricate patterns formed.

7.4 Density effect on spherical simulation

7.5 Number of fibers effect on spherical simulation

Chapter 8

Conclusion

Future directions

-> larger systems -> utilizing multiple GPUs -> problem solving linear system -> memory consumption -> solve linear system without storing matrix -> performance implications