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. Phasellus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Referat

GPU simulering av stela fibrer

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

Contents

List of listings

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.

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

2.1 Stokes flow and Stokeslet

The fluids involved in the rigid fiber simulation are characterized by three properties which allow the general Naviar-Stokes Equations to be immensely simplified.

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* $Re \ll 1$ — The fluid velocities are very slow and/or the viscosity is very large.

Given these constraints and special case of very low Reynolds numbers the general Naviar-Stokes Equations can be linearized to the Stokes Equations

$$\begin{aligned} grad(p) - \mu\Delta\mathbf{u} &= \mathbf{f} \qquad \text{in} \quad \Omega, \\ div(\mathbf{u}) &= 0 \qquad \text{in} \quad \Omega, \end{aligned} \tag{2.1}$$

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.

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 depedence is only reintroduced by boundary conditions. Thus given the boundary conditions the structure of the flow can be calculated.

By taking advantage of the linearity of the Stokes equations, which implies the existence of a Green's function, and introducing boundary conditions so-called fundamental solutions can be found. First no-slip conditions on the surface of the slender bodies are defined as

$$\mathbf{u} = \mathbf{u}_\Gamma \quad \text{on} \quad \Gamma, \tag{2.2}$$

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

Furthermore, the velocity field should be equal to a background velocity $\mathbf{U}_0(\mathbf{x})$ at infinity

$$\mathbf{u} \to \mathbf{U}_0 \quad \text{as} \quad ||\mathbf{u}|| \to \infty. \tag{2.3}$$

Next the force term in (2.1) is replaced by a point force acting at the origin $\mathbf{x}_0$

$$\begin{aligned} grad(p) - \mu\Delta\mathbf{u} &= \mathbf{F} \cdot \delta(\mathbf{x} - \mathbf{x}_0), \\ div(\mathbf{u}) &= 0, \end{aligned} \tag{2.4}$$

where $\delta(\mathbf{x} - \mathbf{x}_0)$ is Dirac delta function. Solving these equations for the velocity field $\mathbf{u}(\mathbf{x})$ yields the fundamental solution

$$\begin{aligned} \mathbf{u}(\mathbf{x}) &= \mathbf{S}(\mathbf{x} - \mathbf{x}_0), \\ \mathbf{S}(\mathbf{r}) &= \frac{1}{8\pi\mu}\left(\frac{\mathbf{I}}{|r|} + \frac{rr}{|r|^3}\right), \end{aligned} \tag{2.5}$$

where $\mathbf{S}$ denotes the Stokeslet tensor and $\mathbf{I}$ the identity tensor. Additionally, we will later need higher order fundamental solutions which can be obtained by simply differentiating the Stokeslet, e.g. the so-called doublet is defined as

$$\mathbf{D}(\mathbf{r}) = \frac{1}{2}\Delta \mathbf{S}(\mathbf{r}) = \frac{1}{8\pi\mu}\left(\frac{\mathbf{I}}{|r|^3} - \frac{3rr}{|r|^5}\right). \tag{2.6}$$

2.2 Slender bodies

2.3 Rigid 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

Chapter 4

Parallel Implementation

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 efficent 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 GPU Programming

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

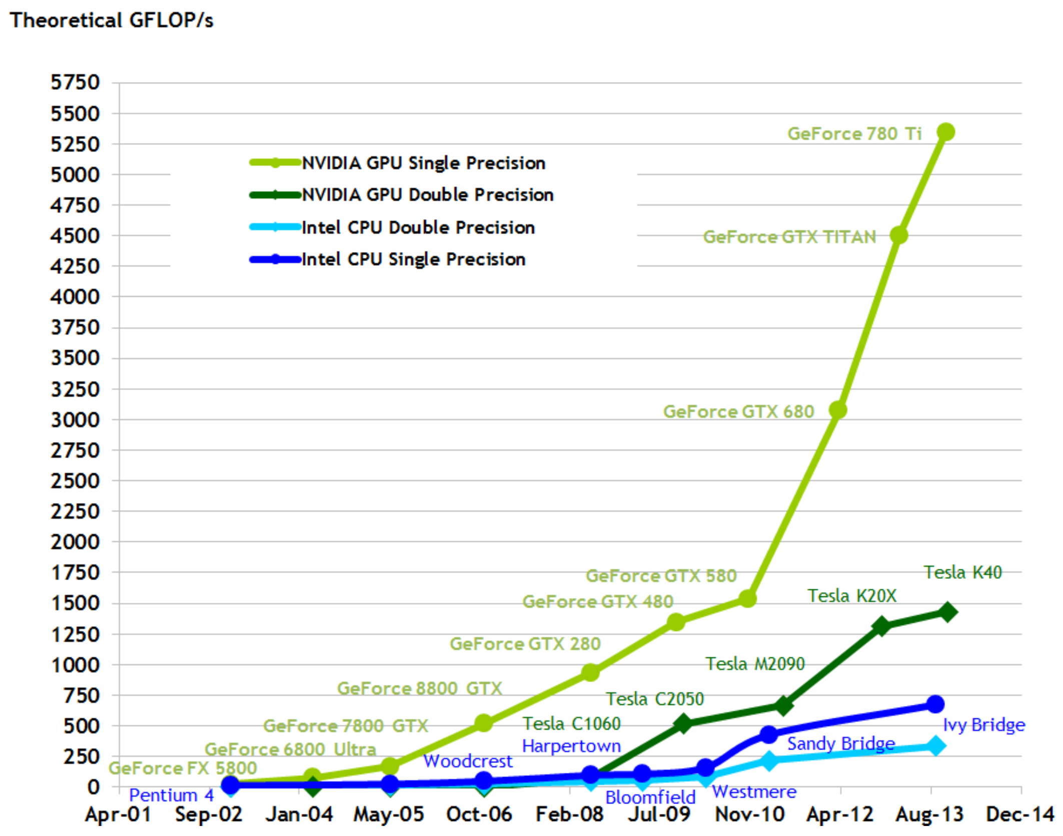


Figure 4.1: Increase in floating-point operations per second for CPUs versus GPUs over time.

general purpose applications. The deciding factor is how well the problem can be parallized 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.

The huge difference in performance mainly derives from the number of independent compute cores. Even though each individual core of a traditional CPU is very fast they usually only have four, eight or maybe sixteen cores. In contrast to this GPUs can have several hundreds of indepedent compute cores. Each core can simulatenously 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 supplimenting 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 implentation 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 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-accerlated 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.

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

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

Listing 4.2.1: Pseudocode for CUDA vector addition

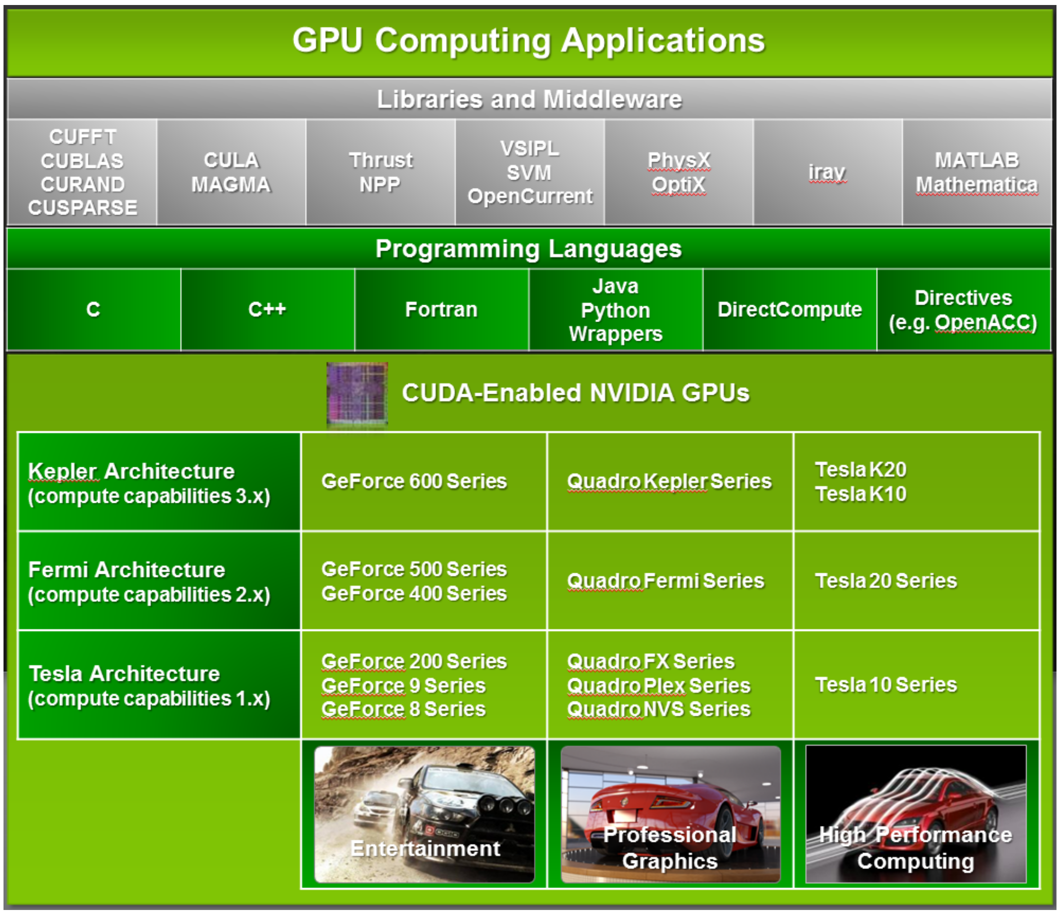

GPU Computing Applications
Libraries and Middleware
CUFFT CUBLAS CURAND CUSPARSE
CULA MAGMA
Thrust NPP
VSIPL SVM OpenCurrent
PhysX OptiX
iray
MATLAB Mathematica
Programming Languages
C
C++
Fortran
Java Python Wrappers
DirectCompute
Directives (e.g. OpenACC)
CUDA-Enabled NVIDIA GPUs
Kepler Architecture (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.

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 hierachy. As discussed previously a GPU consists of many independent compute cores. On Nvidia GPUs theses cores are referred to as Streaming Multiprocessors (SMs). During execution of the application each SM is tasked with running a distinct set of threads. In CUDA these sets of threads are called thread blocks. Each thread block is then distributed to all the available SMs, which allows for automatic scalability depending on the

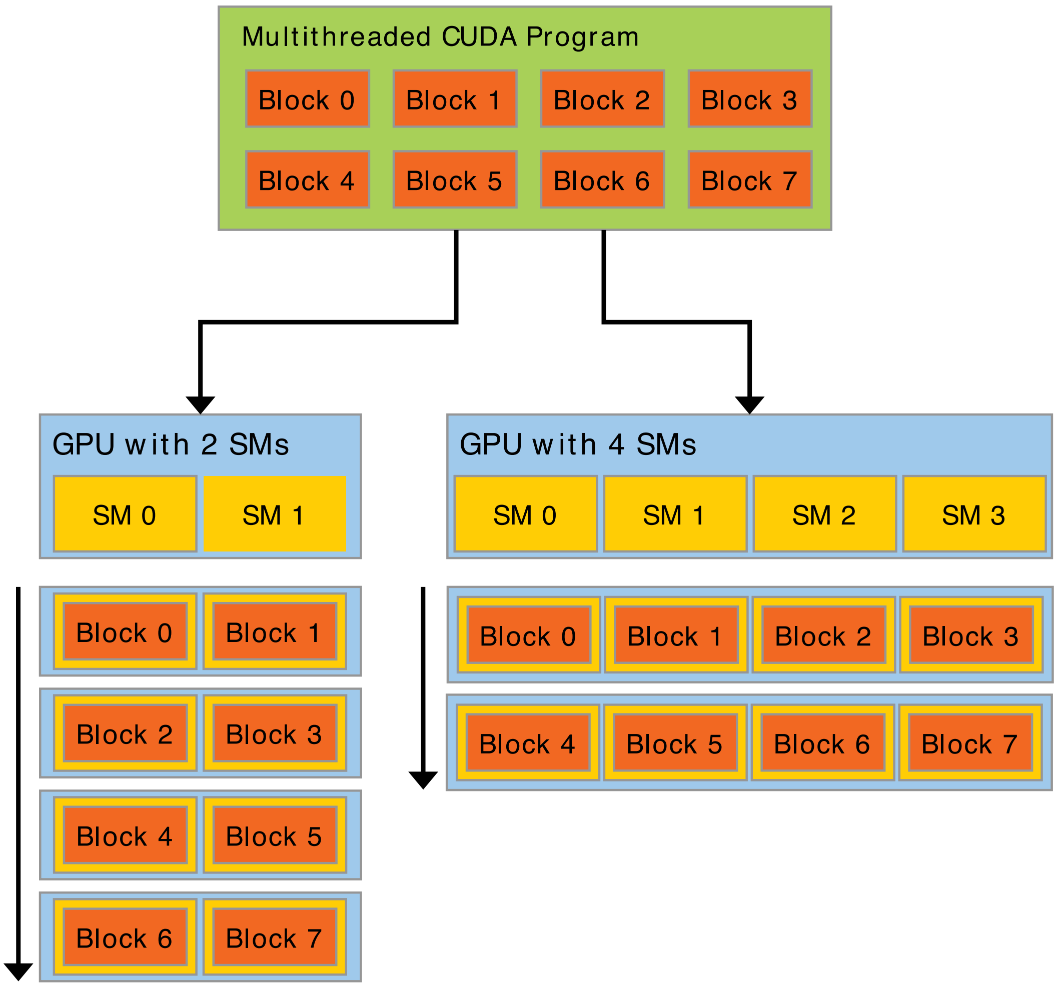


Figure 4.3: Automatic scaling of blocks across an arbitrary number of Streaming Multiprocessors.

number of SMs available as illustrated in figure 4.3.

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 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

```
// 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 4.2.2: 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
- Private per thread

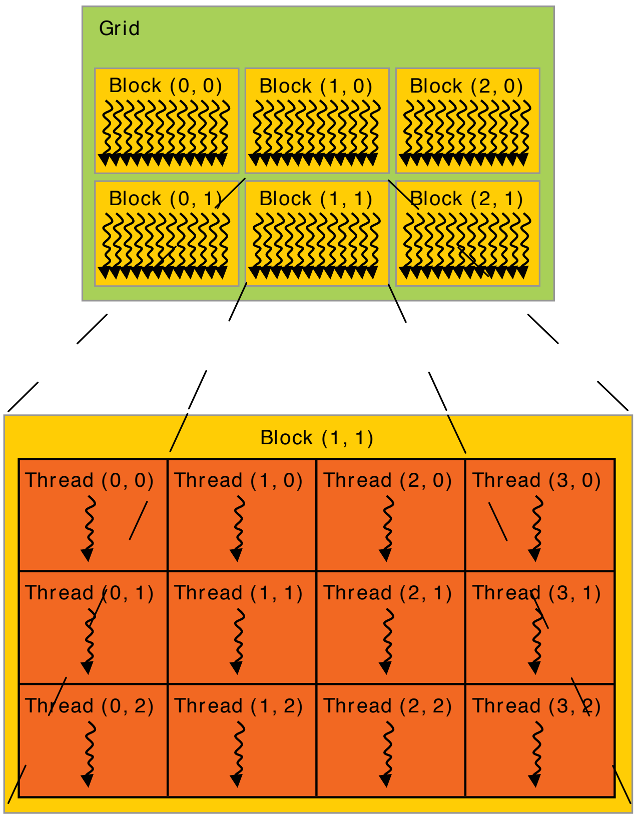


Figure 4.4: Overview of the CUDA platform.

- 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)

4.3 Implementation

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 is 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 Universiy 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. Currentely, 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 where used and tested.

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

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

4.3.1 Kernels

The overall parallel algorithm is very similar to the serial version, however, each simulation step is seperated 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 4.3.1. 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.

The application requires two general configuration files as an input. The first file is referred to as the parameters file which contains the different configura-

```
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 4.3.1: Pseudocode for parallel algorithm on the host.

tion 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 specifiying 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.

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 4.3.2 shows the pseudocode for the one-dimensional implementation of the assemble system step. This means the code is parallelized for each fiber and each thread calculates the contributions to this fiber from all other fibers. Looking at the matrix 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.

```
__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_
    {
      computeInnerIntegral(...);

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

Listing 4.3.2: Pseudocode for the assemble system step with a 1D thread block.

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 coeffcients are used to update the velocities of the fibers. The *Update Velocities* kernel accu-

```
__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; +
  {
    for (int quadrature_index_j = 0; quadrature_index_j < TOTAL_NUMBER_OF_QUADRATURE_POINTS;
    {
      force = computeForce(coefficents, ...)
      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 4.3.3: Pseudocode for the updating velocities simulation step.

mulates 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 4.3.3. 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 using a 2D thread block and the associated performance increases far outweight the potential performance hit of using atomics.

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

  if (i >= NUMBER_OF_FIBERS) return;

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

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

  normalize(next_orientations)
}
```

Listing 4.3.4: 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 4.3.4 implements the second-order multi-step method introduced in section 3.1. As will be seen later in the results in chapter 5 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.

4.3.2 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 5.

Numeric vs. Analytic Integration

The first optimization was already part of the original serial implementation as described in section . 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.

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 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 to 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 than 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.

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 necesseary 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.

4.4 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 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 5

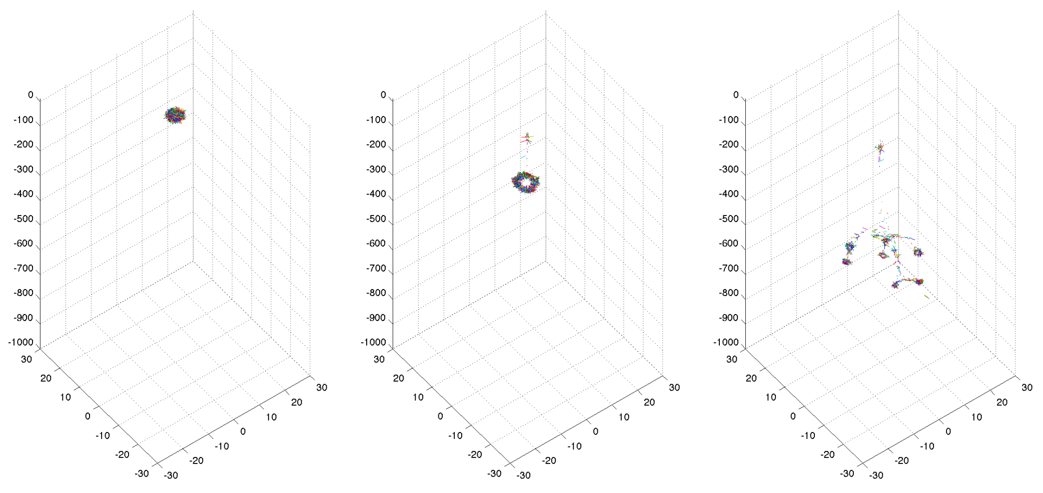
Results

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 optimiziations 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.

5.1 Examples

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 phyiscal phenomenon. Both a run with the fastest CUDA algorithm implemented in the thesis.



(a) $t = 0$ (b) $t = 100$ (c) $t = 285$

Figure 5.1: Visualization Sphere Simulation

5.1.1 Numerical precision

5.1.2 Sphere simulation

5.2 Methodology

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

5.2.1 Hardware

All benchmarks were run on the same workstation with specifications listed in table 5.1. 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.

5.2.2 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

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 5.1: Benchmark system hardware specification

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 seperate 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 configrations 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 than a specficed threshold, the number of iterations is doubled and run again.

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

```
for(int N = 100; N <= 2000; N += 100)
{
int iterations = 4;
while (iterations <= MAX_ITER)
{
for(int i = 0; i < iterations; ++i)
{
generateRandomInitialFiberConfiguration();
run(10); // execute 10 timesteps
collectTimings();
}

rse = calculateRelativeStandardError();

if (rse <= 0.2)
{
break;
}

iterations *= 2
}

reportTimings();
}
```

Listing 5.2.1: Pseudocode for benchmark scheme.

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 5.2.1.

5.3 Optimizations

We now look at the performance results for the different optimizations previously outlined in section 4.3.2. Where applicable the results will be compared between the OpenMP and CUDA version of the algorithm.

5.3.1 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 5.2 illustrates the performance timings for the *Assemble System* step of the parallel OpenMP version. Inline with the observations made by the authors of the

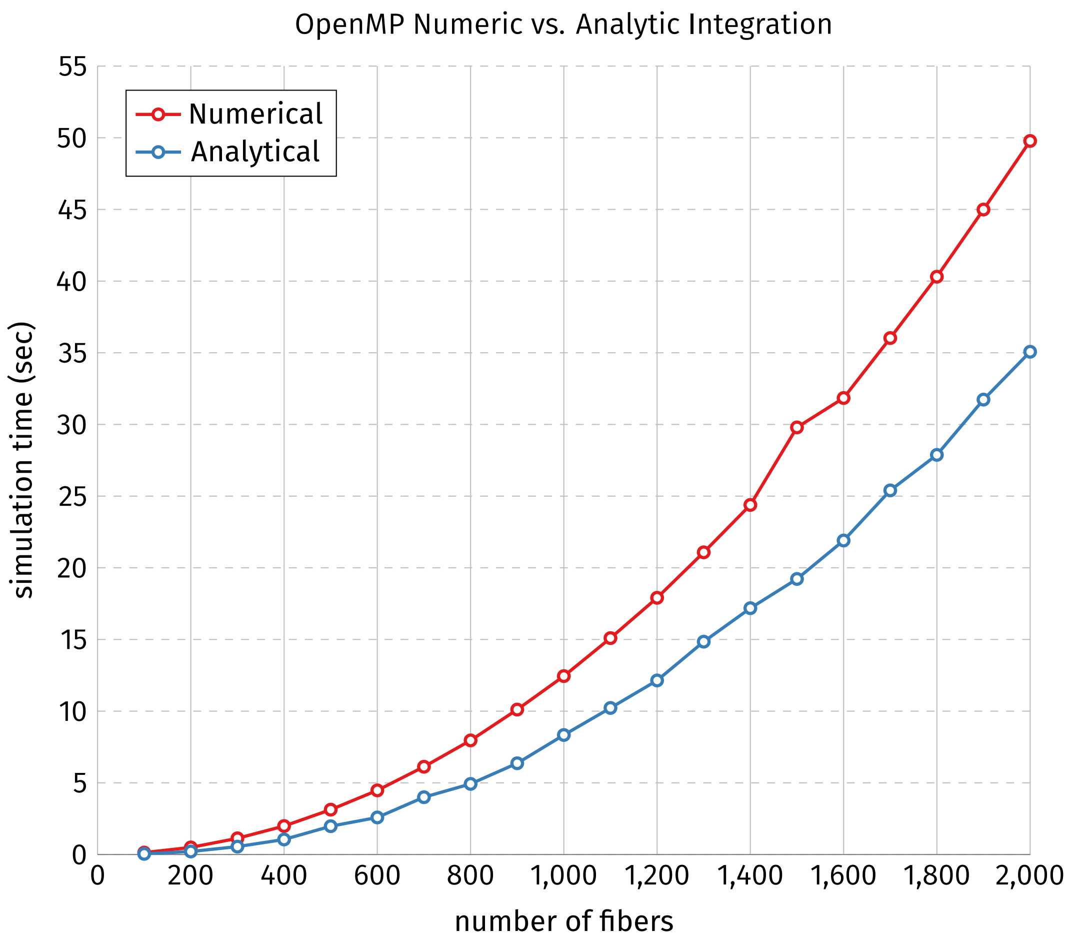


Figure 5.2: Benchmark of assemble system step for integration of inner integral.

original serial implementation the parallel version of the analytic integration is always faster than the numeric integration.

However, the picture is more interesting when we look at the same graph for the CUDA implementation in figure 5.3. Here the results are reversed. The numeric integration outperforms the analytical integration by a large margin increasing with the number of fibers.

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

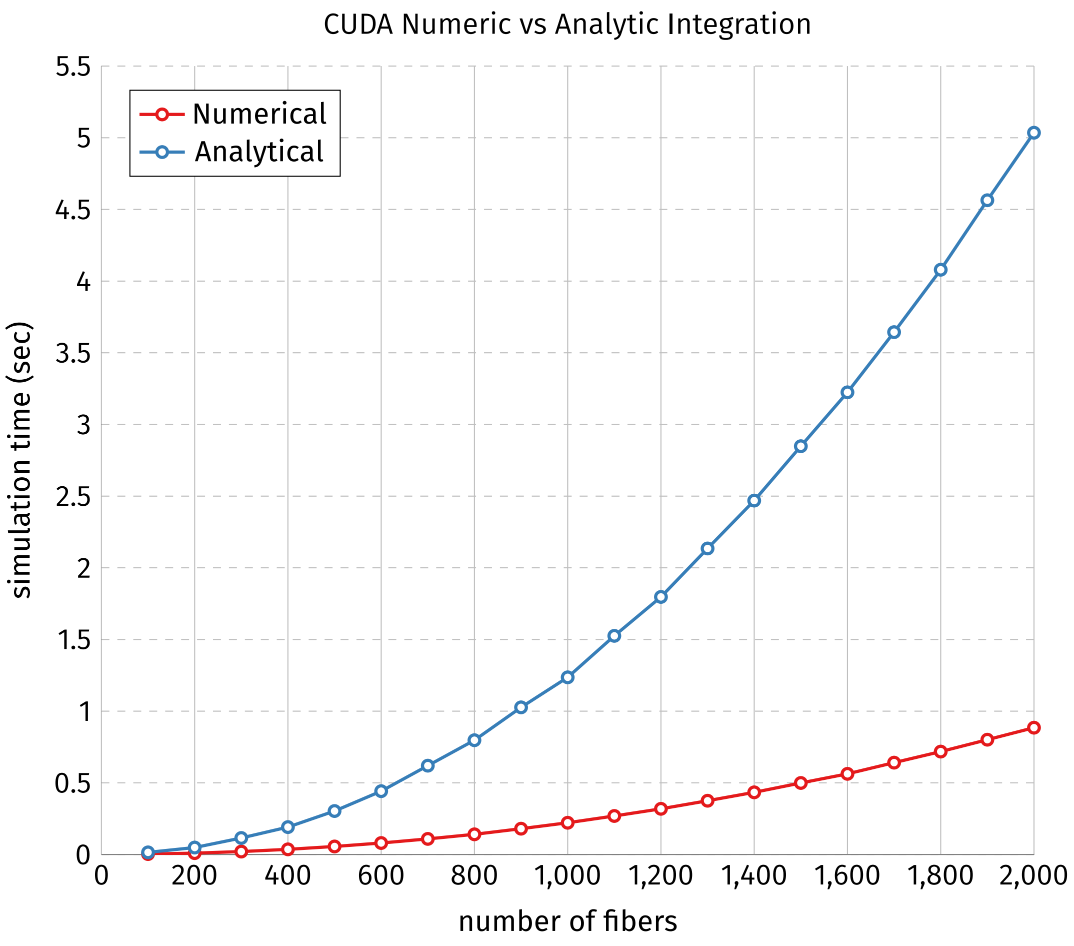


Figure 5.3: Benchmark of assemble system step for integration of inner integral.

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 continous 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 Efficency* shows the ratio of the average active threads per warp to the maximum number of threads per warp. The metrics for both the numerical and analytical integration of the Spherical fiber setup used in section 5.1.2 can be seen in table 5.2.

Algorithm	warp_execution_efficiency
Numerical	99.01%
Analytical	53.79%

Table 5.2: 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 observered performance difference. Closer inspection of the source code reveals that the design constant and the resulting branching as described in section 4.3.2 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.

5.3.2 Shared Memory

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

5.3.3 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 amout of memory writes to the same location and thus the need for potential slow atomics.

The results in figure 5.4 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 be seen in table 5.3, the required atomic transactions for the sphere example from section 5.1.2 are almost two times larger in the 3D case.

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

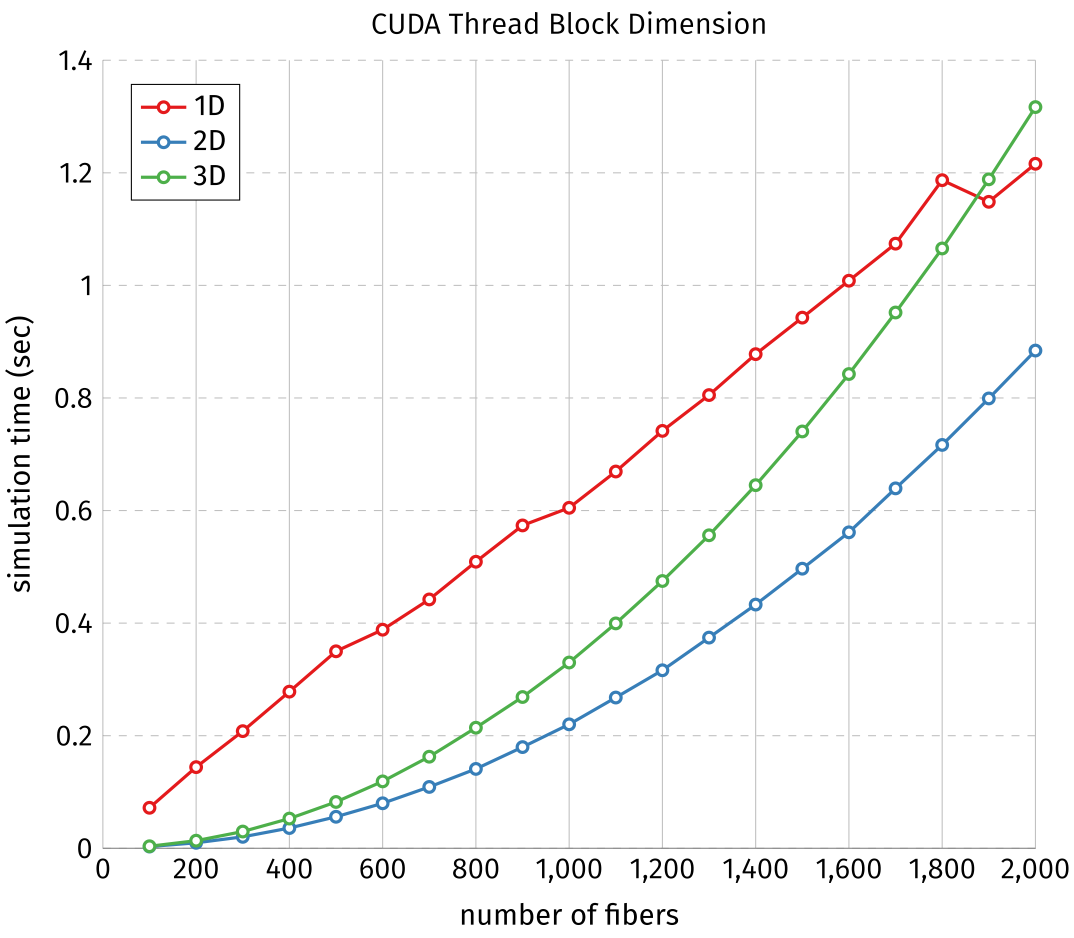


Figure 5.4: Benchmark of assemble system step for different thread block dimensions.

Algorithm	atomic_transactions
2D	1269325
3D	2350670

Table 5.3: Atomic transactions of 2D vs. 3D thread block dimensions.

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

5.4 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 underyling 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 5.5.

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 5.6.

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 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

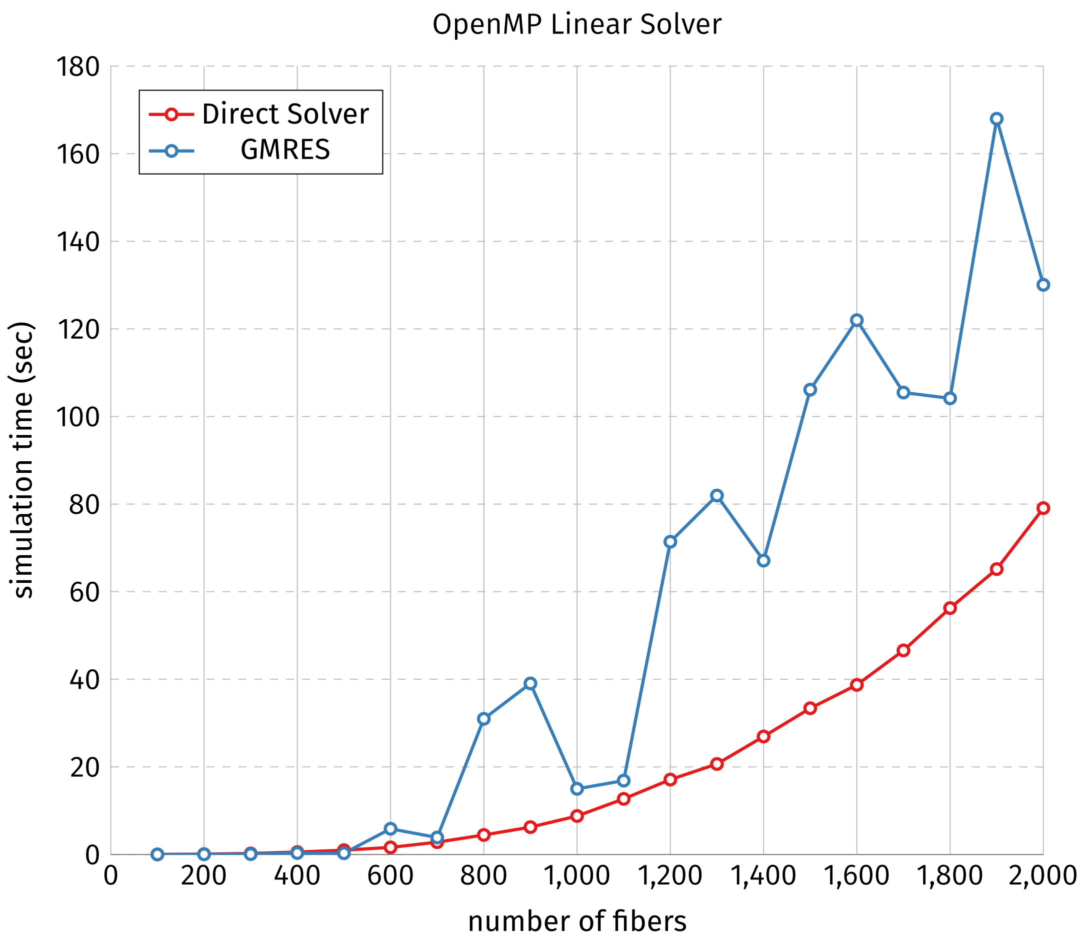


Figure 5.5: Benchmark of solve system step for different Fortran solvers.

is not clear. Either way the direct solver is not only the clear winner in terms of performance but also provides exact solutions.

5.5 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 of the to 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.

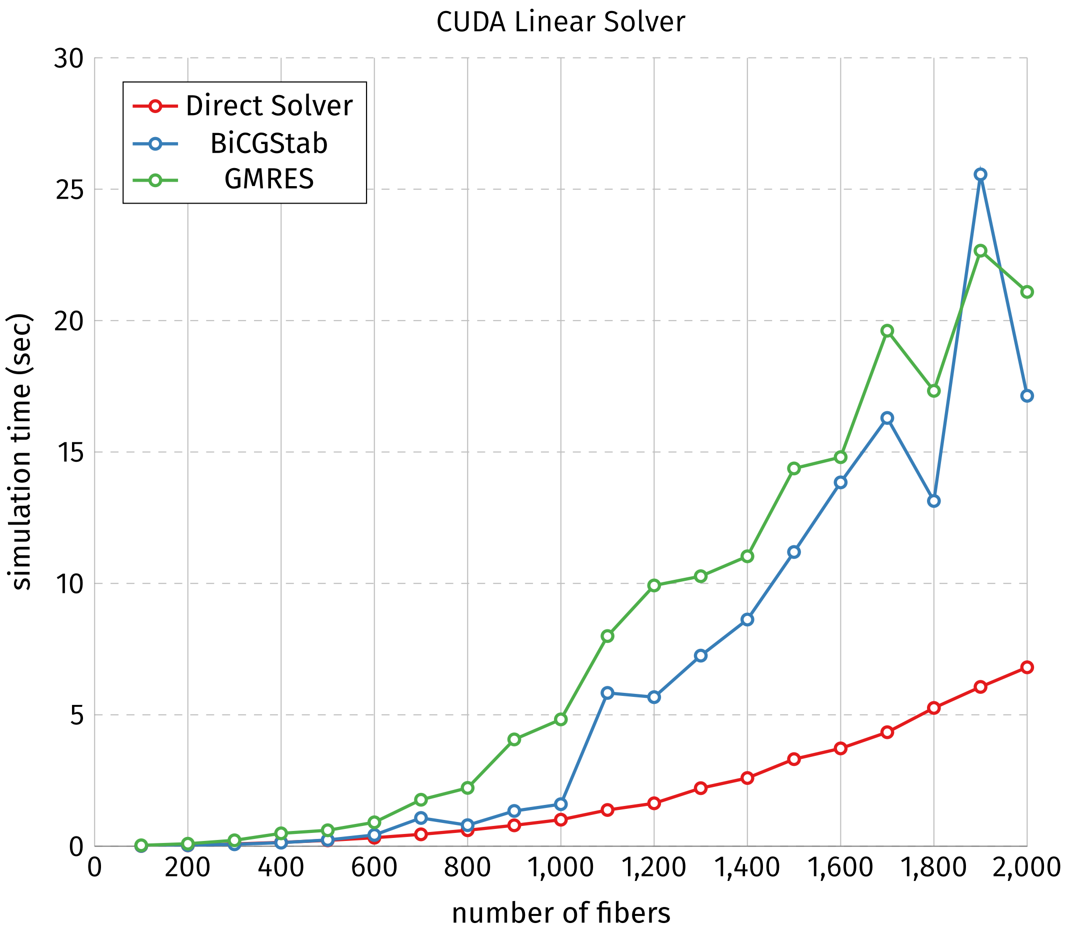


Figure 5.6: Benchmark of solve system step for different GPU linear solvers.

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 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.

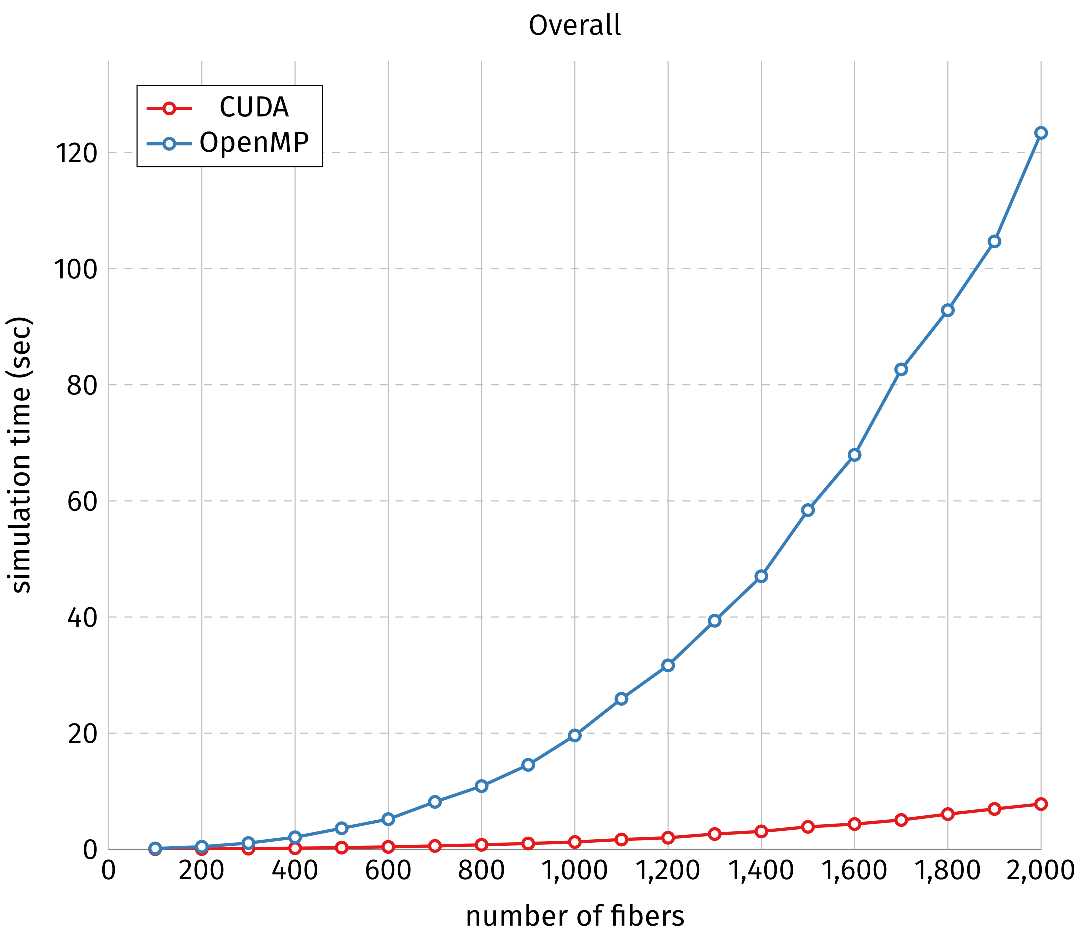


Figure 5.7: Benchmark of overall timestep for both OpenMP and CUDA.

The results for average time required to take a single timestep is illustrated in figure 5.7. 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.

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

We acknowledge that these numbers and performance increases is not necessarily fair. A different CPU and GPU combintation 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× 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 researcoh of rigid fiber simulations.

Chapter 6

Conclusion

Future directions

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