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

In this chapter I will present the required theoretical foundation of the physics behind the simulation. This is required to be able to understand the numerical method used throughout the rest of the thesis.

I will begin by introducing the Stokes flow and its foundatmental solutions as they apply to slender bodies. Afterwards I will focus on the implications this has and how the flow properties can be calulated for the special case of 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 immensly 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 indepdent and time depedence is only reintroduced by boundary conditions. Thus given the boundary conditions the structure of the flow is 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 identy tensor. Additionally we will later need to 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 I 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 I will now introduce the approach used for the numeric simulation. This is crucial to be able to validate the framework against real world experiments.

I 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 final obtained linear system and the shortly 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 implementation of the numerical simulation was discussed. 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. The main focus of this thesis is the implementation on 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 multiple CPUs using the OpenMP framework.

I will begin with a short introducing 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 parallized 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. GPUs can achieve impressive performance increases across a wide range of different applications. The deciding factor is how well the problem can be parallized to take advantage

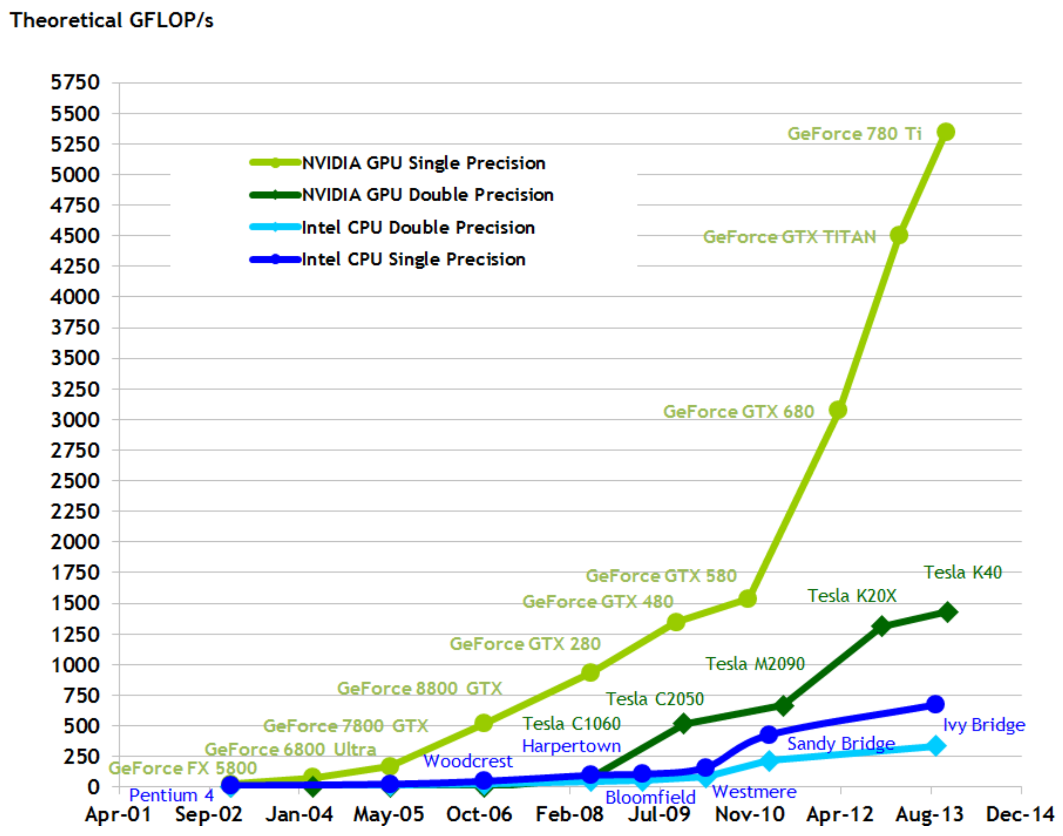

Figure 4.1: Increase in floating-point operations per second for CPUs versus GPUs over time.

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.

In contrast to traditional CPUs which have four, eight and sometimes very fast compute cores, GPU can have many hundreds of independt 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 with more and more of them either supplimenting GPUs or even exclusively relying on GPUs for their computations.

In order to take advantage of these new massively parallel architectures new Application Programming Interfaces 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 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 many different devices. Both Intel and AMD provide the API for their processors and both AMD and Nvidia have drivers available for their GPUs. However this advantage can actually also be a disadvantage as the achievable performance might suffer from the abstraction across all these different kinds of devices being not optimized for a particulars device specific architecture. CUDA on the other hand is in theory highly optimized to achieve the best possible performance on Nvidia's GPUs. In practice the difference might be able to be mitigated by spending the extra time to fine tune the OpenCL implementation to the hardware specific needs. Another disadvantage of OpenCL is the potentially outdated and inconsistent driver support for the various devices. This is especially true for Nvidia which 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 speciliazed setup and simulation scenario. The application will not be widely distributed and only used for internal purposes.

4.2 CUDA

The Compute Unified Device Architecture (CUDA) 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 varity of different options to program the GPU. The easiest way is to simply link to any CUDA-accerlated library and simply using the libraries interfaces from any software environment. For

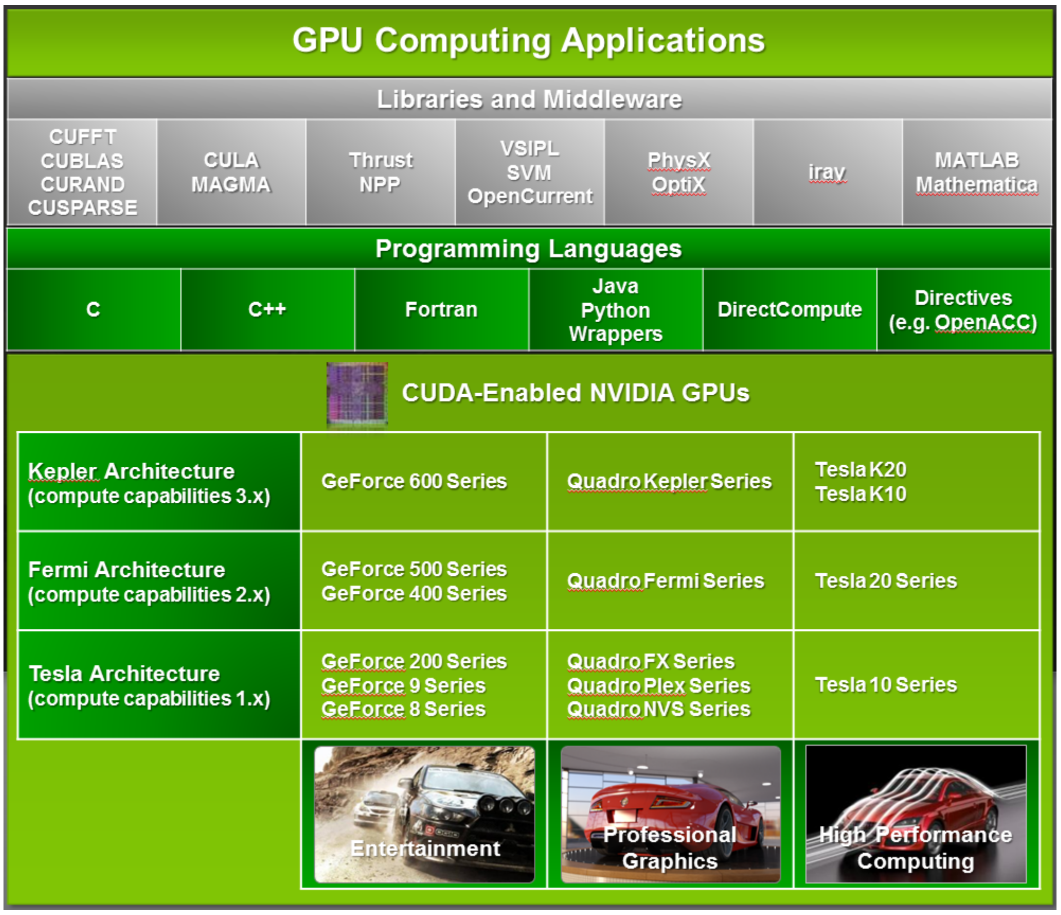
GPU Computing Applications
Libraries and Middleware
CUFFT CUBLAS CURAND CUSPARSE
CULA MAGMA
Thrust NPP
VSIPL SVM OpenCurrent
PhysX OptiX
iray
MATLAB Mathematica
Programming Languages
C
C++
Fortran
Java Python Wrappers
DirectCompute
Directives (e.g. OpenACC)
CUDA-Enabled NVIDIA GPUs
Kepler Architecture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

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 integrating into whatever 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 so called kernels. CUDA kernels are the equivalent of normal C functions, however 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 see the code sample in Figure 4.2.1 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

CUDA Kernels It is important to remember that each kernel invocation is executed independently and no ordering is guaraentedd. It is therefore essential to make sure to avoid any race conditions or shared memory access. There are 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 indepent 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 number of SMs available on the specfic GPU device as illustrated in figure 4.3.

Thus the developer only has to divide the workload into appropiately sized blocks of threads and invoce 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 implementation and type of work being done. In practice the size is often chosen by running benchmarks with various different size to determine the sweet spot.

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

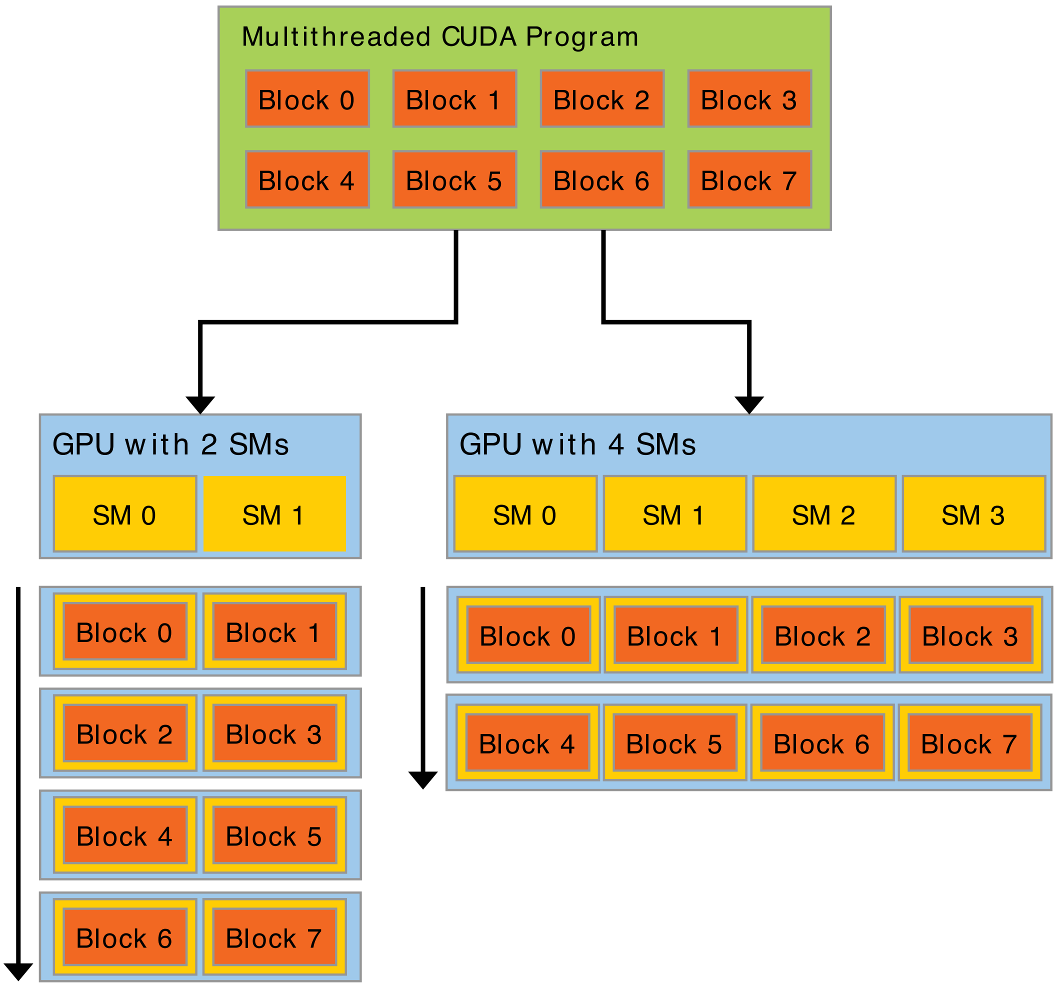

Figure 4.3: Automatic scaling of blocks across an arbitrary number of Streaming Multiprocessors.

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 parallezing 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

Finally because the resources of each Streaming Multiprocessor are limited there exists a upper bound of how many threads a block can contain. Currently is 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.

```
// 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

Grid organize 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 illustrates an example configuration of a 2D grid with 2D blocks.

Memory hierarchy In addition to the Thread hiearchy...

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

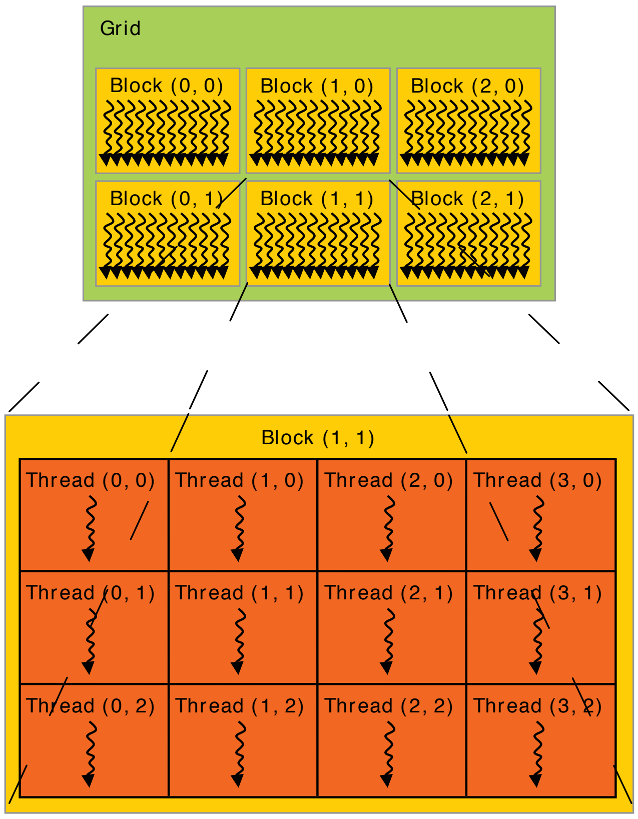

Figure 4.4: Overview of the CUDA platform.

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 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 parallize 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 enviroment used in the later experiemntal usage of the resulting application. The build system for compiling and linking the final application was CMake, as it is a widely used open-source and cross-platform build system, allowing for easy integration of the various required libraries in a well documented and straightforward manner.

Under the hood the build system used Nvidia's CUDA platform tools to compile the code using *nvcc*, the Nvidia's LLVM-based CUDA compiler. In addition to the CUDA libraries 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 provides the implementation for the direct solver used during this thesis. This dense linear algebra library provides features similar to standard LAPACK functions but for multicore architectures. It also provides features to support hybrid algorithms across multiple different archictures, however these features where 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 provided 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 makes sense. Thus a CPU based BLAS implementation is also needed. For this I chose the OpenBLAS library 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 parallization methods in order to provide 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 support OpenMP, OpenCL and most importantanly for this thesis CUDA.

While mostly focussed on sparse matrices for the implemented iterative solvers, ViennaCL also support solving dense matrices using a variety of different iterative solvers. As the rigid fiber simulation exclusive 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 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 simply 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 variantions 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 run 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 invocate in a serial manner, this means CUDA guarantess that all data modified in a kernel is available before the next kernel is executed. These kernels are than 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 algorithm is illustrated in listing 4.3.1.

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

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 constant 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

Each of the parallized substeps are now discussed in more detail. I will discuss the purpose of each kernel as well as the required input and outputs.

```
__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.

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 parallized for each fiber and each thread calculates the contributions to this fiber form 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 the each fibers and its orientation. Using these combined with the equations outlined in chapter 2 and chapter **??** 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 the subsequent steps.

Update Velocities After solving the linear system the solution coeffficents are used to update the velocities of the fibers. The *Update Velocities* kernel accumulates the excerted 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 this would result in undefined behaviour and incorret 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, guararenteeing 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 show that using a 2D thread block and the associated performance increases for outweight the potential performance hit of using atomics.

```
__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.

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 **??**. As seen later in the results in chapter ?? this required time for this kernel is minuscule compared to the other steps. The kernel only scales linearly and additional has a perfectly aligned memory access resulting in close to optimal usage of the GPU hardware.

4.3.2 Optimizations

- Great care taken to optimize code through development
- Small code-level optimizations, allocating variables, unneceassy copy operations

```
__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.

- More advanced optimizaiton loop consolidation, loop unrolling
- Contionous use of benchmark suite to avoid regressions
- Many optimizations are both applicable to CPU and GPU
- However some result in very different result
- Introduce optimizations
- Performance results later in results section

Numeric vs. Analytic Integration

Shared Memory

Thread Block Dimension

4.4 OpenMP

- Fairer comparison between CPU and GPU -> parallized CPU implementation - What is OpenMP - Porting process - No difference for parallization dimension

Chapter 5

Results

The last chapter introduced the parallel implementation of the numerical simulation. It introduced the concept of general purpose computing on modern GPUs as well as giving a practical overview of the implementation of the algorithm using nVidia CUDA framework and possible optimiziations to take advantage of the unique properties of the GPU architecture.

Using all the available implementations of the algorithm this chapter will showcase a multitude of different result 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 Sphere simulation

5.2 Methodology

5.2.1 Hardware

5.2.2 Benchmark setup

5.3 Optimizations

5.3.1 Numeric vs. Analytic Integration

5.3.2 Shared Memory

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

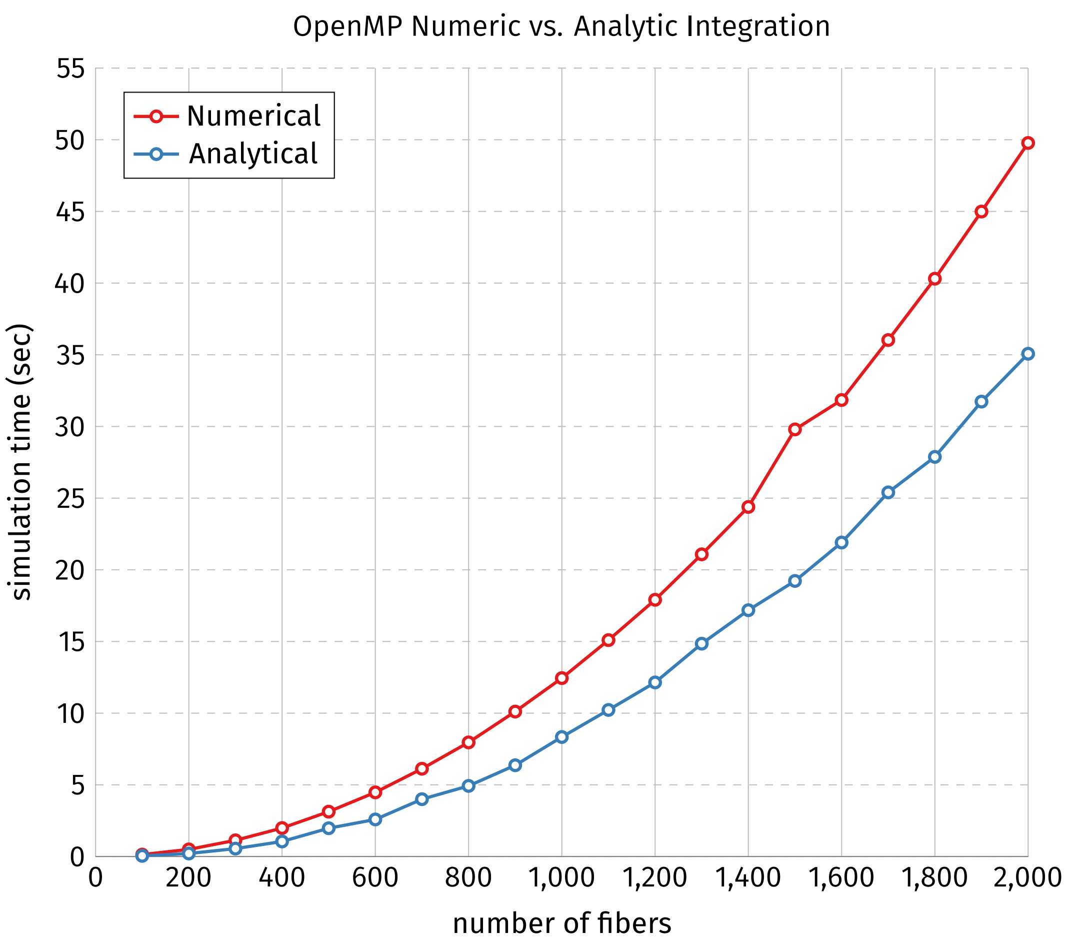

Figure 5.1: Benchmark of assemble system step for integration of inner integral.

5.3.3 Thread Block Dimension

5.4 Linear Solvers

5.5 Comparing CPU and GPU performance

5.6 Fortran vs. CUDA

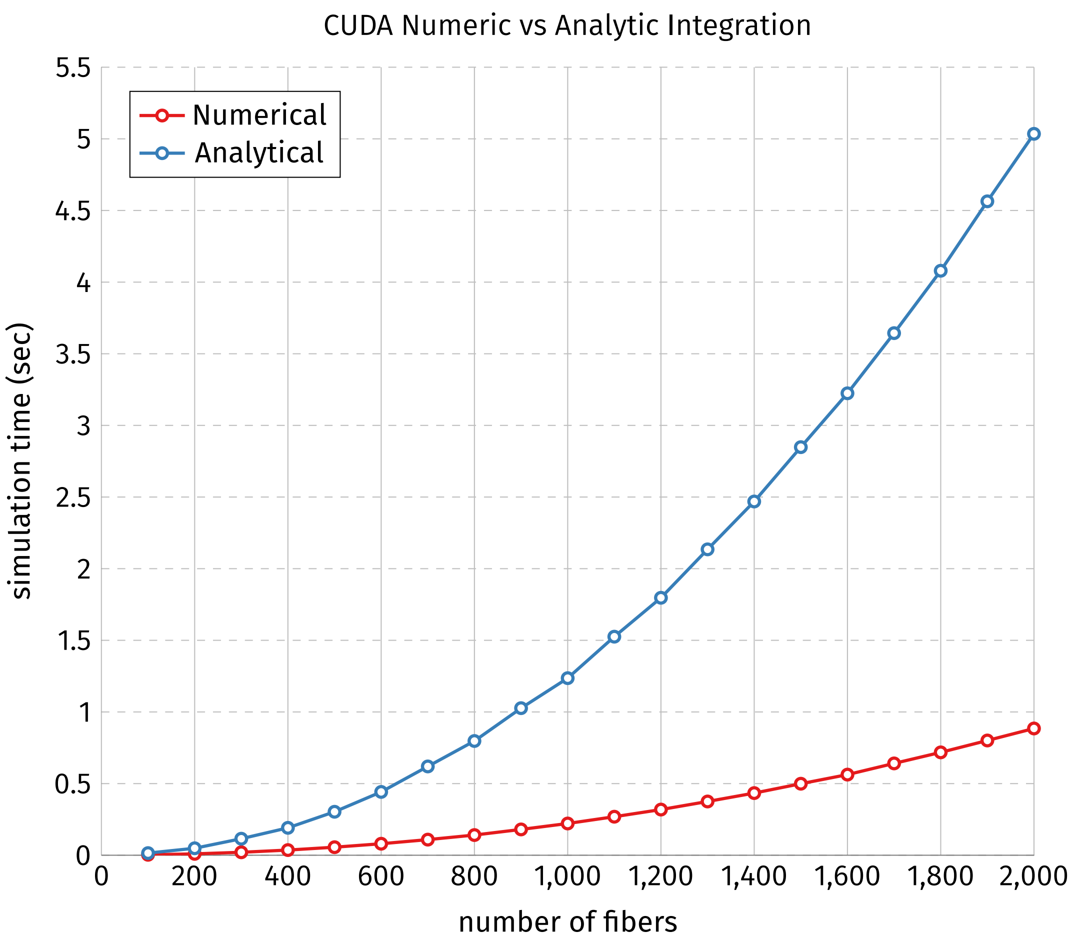

Figure 5.2: Benchmark of assemble system step for integration of inner integral.

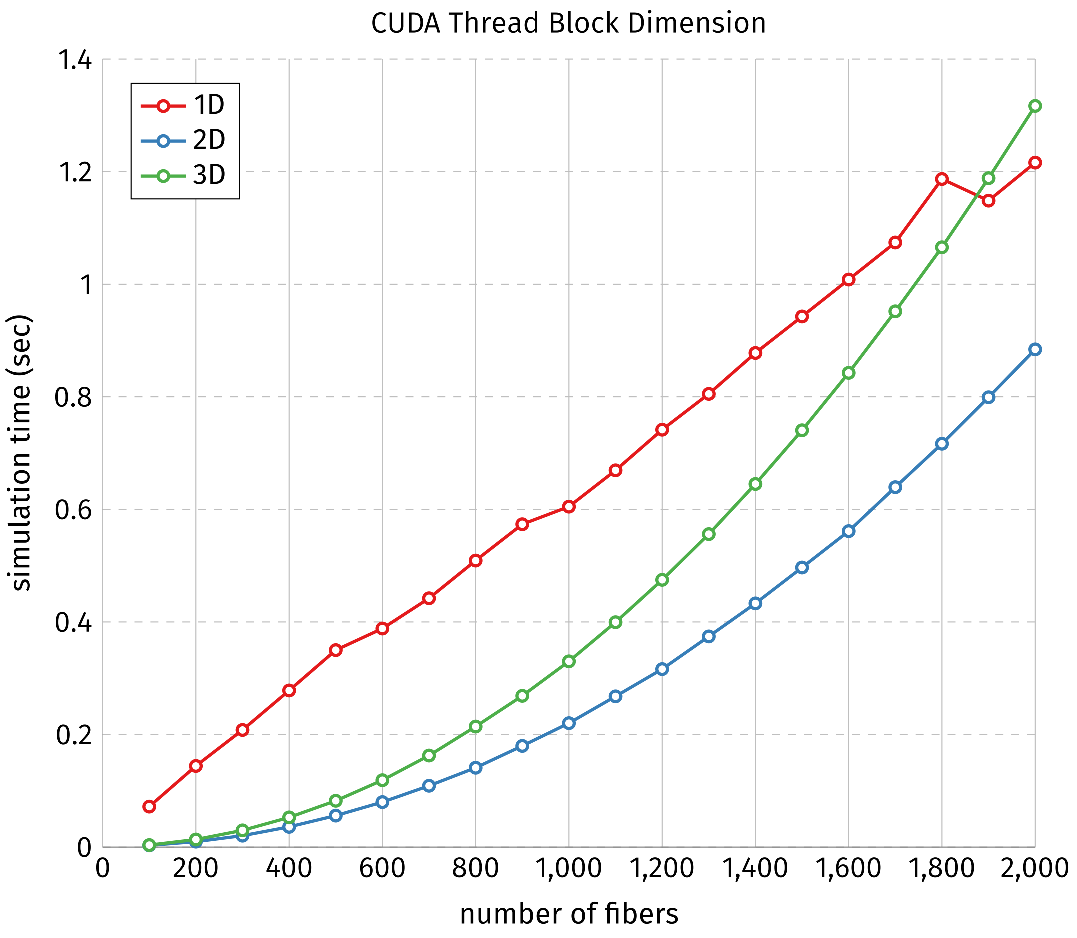

Figure 5.3: Benchmark of assemble system step for different thread block dimensions.

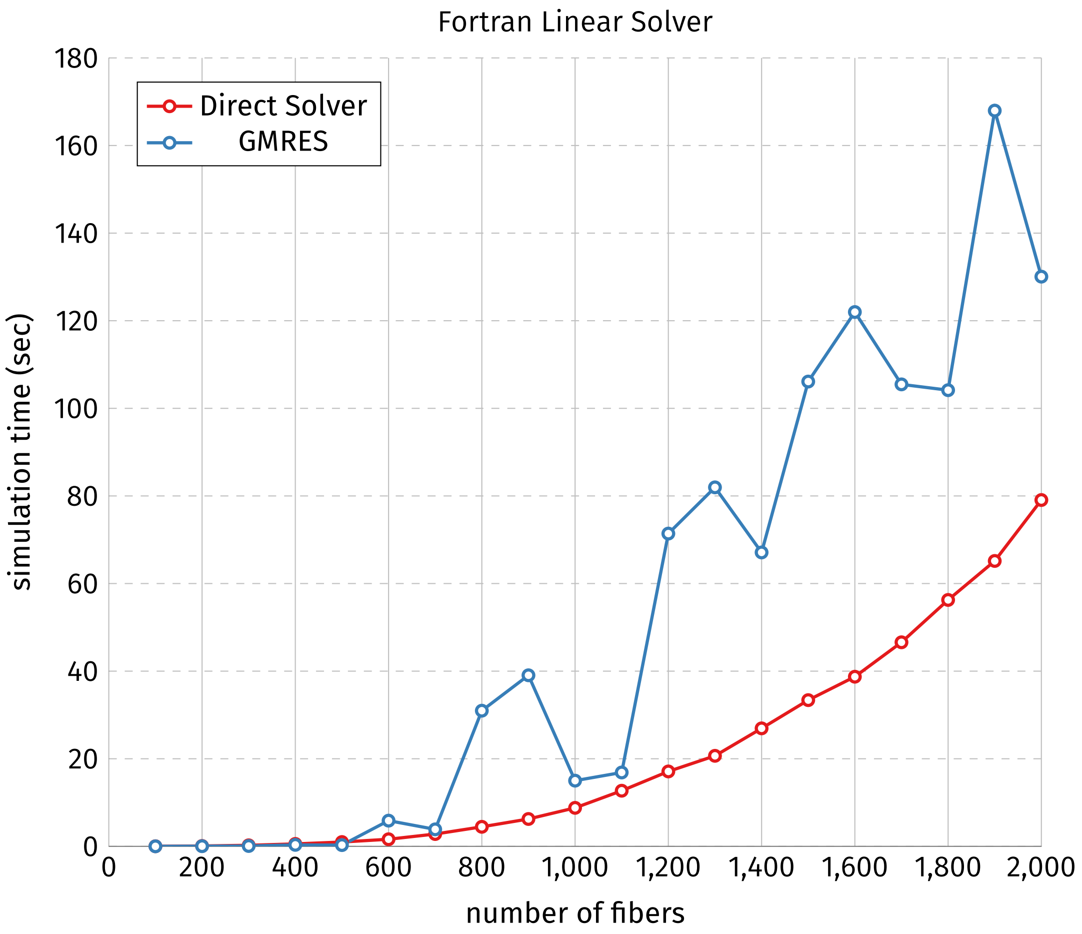

Figure 5.4: Benchmark of solve system step for different Fortran solvers.

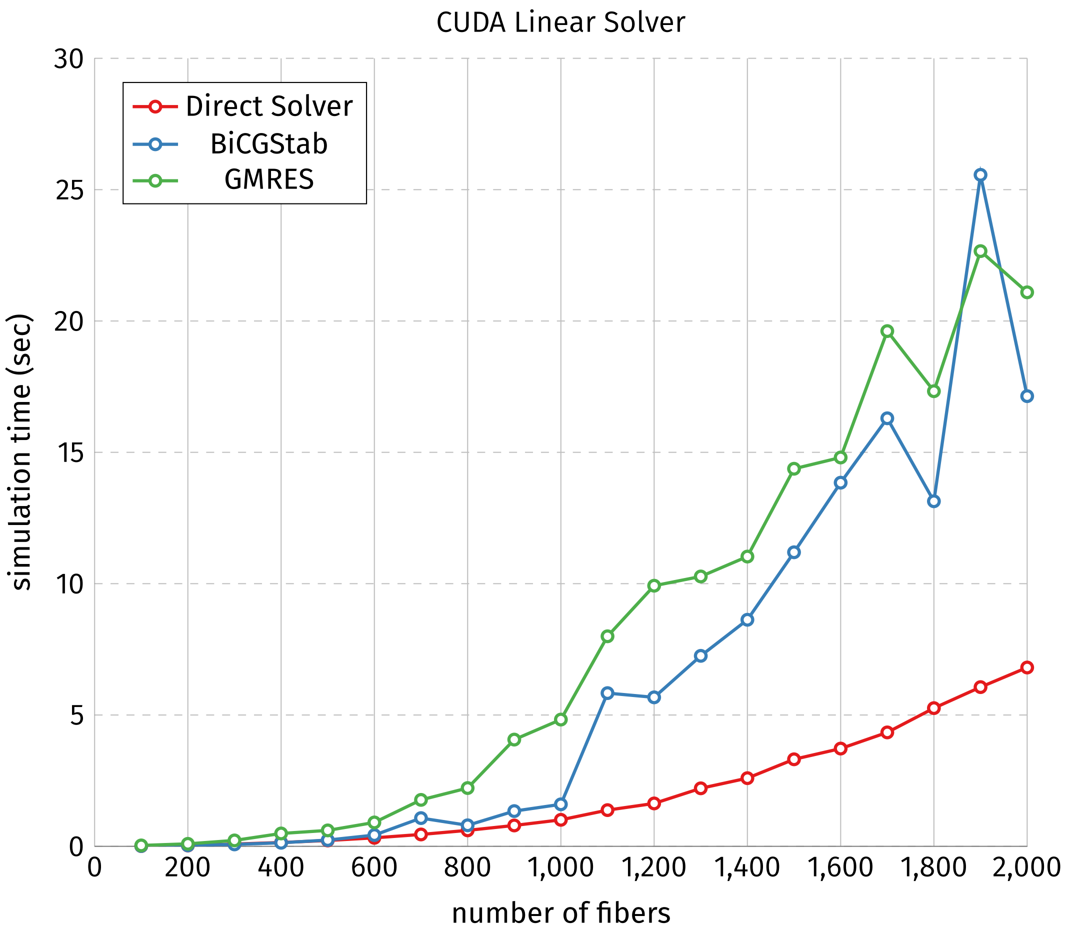

Figure 5.5: Benchmark of solve system step for different GPU linear solvers.

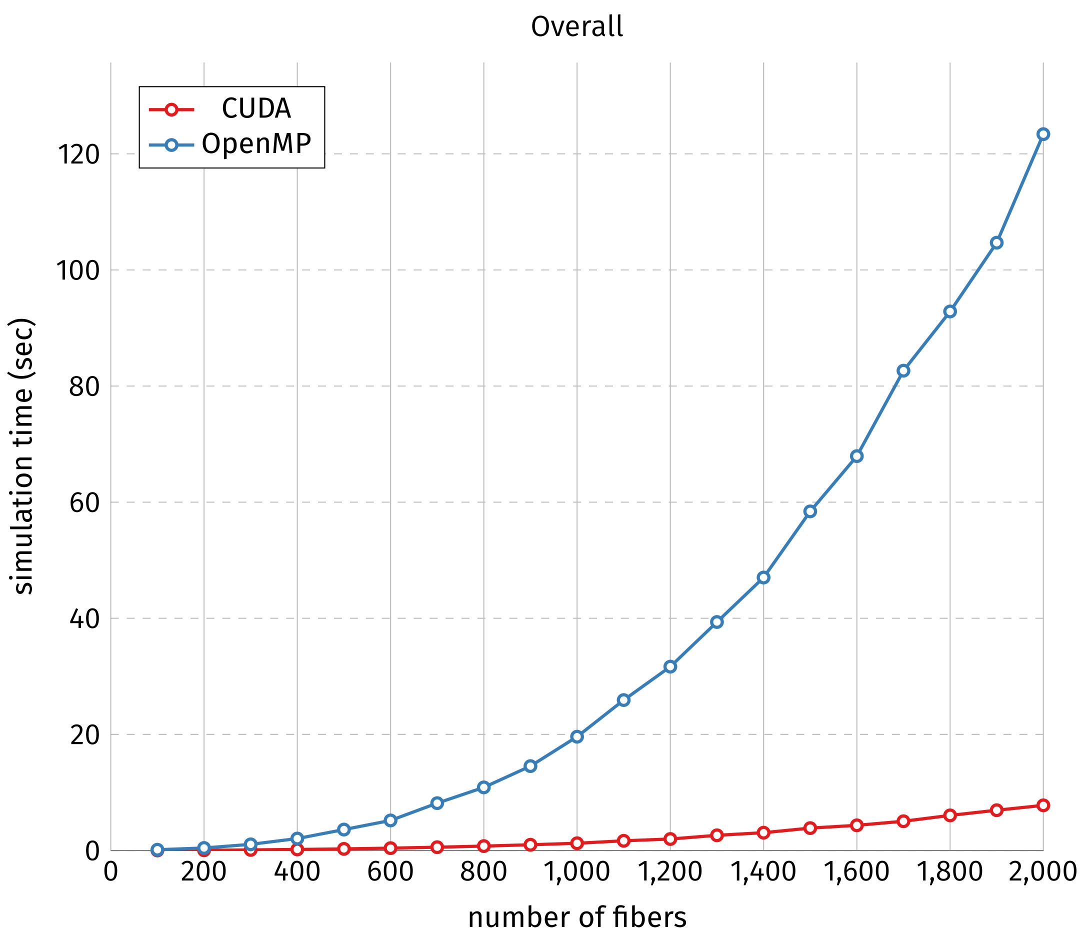

Figure 5.6: Benchmark of overall timestep for both OpenMP and CUDA.

Chapter 6

Conclusion

Future directions

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