UPPSALA
UNIVERSITET

IT 14 029
Examensarbete 15 hp
Juni 2014

GPU-Parallel simulation of rigid fibers in Stokes flow

Ronny Eriksson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

GPU-Parallel simulation of rigid fibers in Stokes flow

Ronny Eriksson

The simulation of a fiber suspension requires that all interactions between the fibers involved are computed. This is a compute-intensive N-body problem that is highly data parallel. Using the GPU for these types of computations can be very beneficial. In this thesis an extension to a simulator, written in MATLAB, for rigid fibers in Stokes flow was designed and implemented to improve the performance of the simulator. A part of the simulator responsible for computing these fiber-to-fiber interactions was ported to the GPU using the CUDA programming language dedicated to general-purpose computing on GPUs. To accomplish this an interface to MATLAB was created and the portion of code to be ported was parallelized and adapted in a way suitable to the GPU. The ported code proved to be 16 times faster than the original implementation.

Handledare: Stefan Engblom
Ämnesgranskare: Jarmo Rantakokko
Examinator: Olle Gällmo
IT 14 029
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Numerical simulations can provide ways to study physical processes that are hard to observe experimentally and also give the ability to completely control the variables that define the state of a system. In the case of rigid fiber suspensions, the rheological properties of these flows are not completely understood. This process plays a central part in a number of practical applications making it an important subject to understand. Some examples are paper-making where it is desirable to have a homogenous fiber distribution and wastewater purification where the sedimentation behaviour is of interest.

Properties such as the settling velocity of a fiber depend on the orientation of the fiber and the formation of fiber clusters [15]. The ability to study the effects of individual fiber interactions can give valuable information on this process, but are hard to isolate in a real suspension. While this is easily accomplished with a simulator, the complex interactions between each fiber combined with the large number of fibers involved is a challenging process to simulate both efficiently and accurately.

The performance can be improved by a number of different approaches. One approach is to modify the numerical model to make it less compute-intensive. This is not always possible without some sacrifice to the accuracy and could be unsuitable in a simulation of a physical process. A preferable approach is to parallelize the existing model. The advantage of a parallelization is that, ideally, the numerical model can be left untouched and still achieve better performance. The final gain of a parallelization depends both on how parallelizable the algorithm is and on the hardware used. A parallelization requires that the algorithm is divisible into smaller parts that can run concurrently and the hardware impose additional requirements on each part for the parallelization to achieve the best performance.

With application programming interfaces (APIs) like Pthreads [13] and OpenMP [14], an application can be divided into threads that are executed concurrently on the cores of a multi-core CPU. As general-purpose computing on graphics processing units (GPGPU) has become increasingly popular with support from AMD, Intel and NVIDIA, the GPU has also become a viable option along with the CPU as a platform for parallel computing implementations. A GPU has a much larger number of cores than a CPU but each core is on the other hand more limited. Since a multi-core CPU and a GPU have different requirements, a parallel algorithm that performs well on a CPU will not necessarily perform well on a GPU. The former is more lenient than the latter but a GPU implementation can on the other hand be up to two orders of magnitude more efficient [8].

A pre-existing simulator for rigid fibers in Stokes flow will be presented in the next chapter. A compute intensive part of the simulator has been identified to have characteristics that are suitable for a GPU. The most important characteristic is the possibility to split the algorithm into a large number of independent computations. This type of algorithm usually maps well to the large number of computational units available on the GPU and the potential performance is therefore higher using a GPU as opposed to a CPU. The focus of this thesis will be on an extension to the simulator where a GPU parallelization of this compute intensive part is designed.

2 Accurate slender fiber dynamics

2.1 Overview of FIBR

FIBR is the name of the simulator that will be extended. It simulates the behaviour of a group of fibers in three-dimensional space that sediment in a viscous fluid. A numerical method with an emphasis on accuracy is used that takes advantage of a number of properties concerning the fibers and the fluid [15, 1]:

- The external forces affecting the system are assumed to be known in advance.
- Each fiber is a microscopic rigid fiber with a rigidness that ensure that no deformation will occur due to any forces acting on the system.
- Each fiber is slender, straight and has an equal length.
- The fibers have a non-Brownian behaviour with a rigid body motion where they can rotate and translate.

Due to the viscosity of the fluid, the flow is being treated as a Stokes flow and the fibers are discretized by placing fundamental solutions to the Stokes equations along the center-line of each fiber. The accuracy of the final result can be adjusted through a number of parameters that are set before the simulation.

It is possible to randomly distribute each fiber inside a volume of choice before the simulation begins. This is a convenient way to initialize each fiber position and will be taken advantage of for all simulations carried out. The volume of fibers can be considered to be in free space by the simulator or replicated in space using periodic boundary conditions. This thesis will consider the free space part of the simulator.

The duration of the sedimentation that is simulated is divided into a series of time steps of equal size. Assuming that the gravity is the only external force and the fiber is aligned with the gravity; one unit of time is crafted so that it equals the duration it takes for one fiber to sediment half its length [15]. The final result of the simulation can be presented graphically in the form of plots for each successive time step.

It is preferable to use a large number of fibers since the behaviour of the fibers on a macroscopic scale becomes more realistic as the number of fibers increases. The simulator contains an N-body problem, where all interactions between N points are calculated. The number of points depends on the number of fibers, but also on the accuracy chosen before the beginning of the simulation. This together with the number of time steps that are needed form a conflict between the accuracy and the performance. Improving the performance can therefore increase the feasibility of more realistic simulations.

2.2 Matrix Laboratory

FIBR is largely implemented in Matrix Laboratory (MATLAB) with the exception of some optimized functions made in the C programming language. MATLAB is both an integrated development environment (IDE) and a high-level programming language which

focus on numerical computations [5]. The language is an interpreted language with a concise syntax for operations involving matrices and it also has an automatic memory management. MATLAB is popular in academia and science and comes with a large library of numerical functions. Some of these functions also have a built-in support for multiple threads.

While this simplifies the implementation of algorithms, the convenience can be a drawback for some performance critical algorithms where MATLAB is too slow. It could therefore be preferable to implement them in a lower-level language such as the C/C++ programming language.

2.2.1 MATLAB executable

The extension to FIBR will be implemented using the CUDA programming language. Since FIBR is implemented in MATLAB, a way to call CUDA from within MATLAB is therefore needed. This functionality can be provided by MATLAB executable (MEX) [6]. A MEX-file is a C/C++ or Fortran source file that works as a bridge between MATLAB and another language. A special function is required to be included and is used as the entry point to the MEX-file. This function provides the interface understood by MATLAB and makes it possible to communicate data with MATLAB. The final MEX-file is compiled into a dynamically linked shared library and is callable from MATLAB in the same way as a native function.

A number of things need to be acknowledged before implementing a MEX-function. Because of MATLABs internal memory management, some care must be taken when dynamically allocating memory. If the memory is passed to MATLAB from the MEX-function it needs to be allocated using allocation functions provided by MATLAB. It is also recommended to do this for memory used within the MEX-file so that MATLAB can free the memory in case the MEX-file crashes before the allocated memory is freed. Since a MEX-function is essentially a shared library, it is loaded into memory when it is first called from MATLAB and it will stay there until it is manually or automatically cleared from memory. When it should be cleared from memory is up to MATLAB to decide, which needs to be taken into consideration if the MEX-function maintains a state between successive calls. A function can be registered to be called before the MEX-function is cleared so that a proper cleanup can be performed if needed.

2.3 Current bottlenecks

A profiler is included in the MATLAB IDE that will be used in this section to time the different functions that constitute the FIBR simulator. A profiler measures how much time is spent in different parts of a program during the execution. Some information that is provided for each function is:

- The number of calls to the function and to each individual line of code inside the function.
- The total time spent inside the function as well as in each individual line of code.

- The self-time inside the function, which is the total time excluding the time spent inside any child functions.

The amount of detail for a MEX-function is restricted to the number of calls to the function and the total time spent inside the function. No further analysis of the internals of a MEX-function is provided by the MATLAB profiler. This does however not pose a problem for the current analysis.

2.3.1 Initial profiles

The profiles for two different simulations with 300 and 600 fibers are summarized in Table 2.1 and 2.2 respectively. All parameters were kept at their default value except for the number of fibers that was explicitly given. The time frame for both simulations was set from 0 to 6 time units with a time step of 0.1. From the tables alone we can see that, excluding the time spent inside any child functions, most time is spent inside `velocity` and `greenint`.

Function Name	**Calls**	**Total Time**	**Self Time**
advect	1	540.736 s	0.019 s
velocity	61	540.594 s	264.615 s
assemble	61	275.979 s	61.434 s
greendotleg1	18300	179.607 s	28.038 s
greenint	18300	100.836 s	100.836 s
tprod	237960	82.298 s	82.298 s
tsum	18483	3.372 s	3.372 s

Table 2.1: Summary of profile - 300 fibers

Function Name	**Calls**	**Total Time**	**Self Time**
advect	1	2344.284 s	0.013 s
velocity	61	2344.218 s	1446.054 s
assemble	61	898.164 s	176.282 s
greendotleg1	36600	608.394 s	109.045 s
greenint	36600	337.817 s	337.817 s
tprod	475860	261.215 s	261.215 s
tsum	36783	13.805 s	13.805 s

Table 2.2: Summary of profile - 600 fibers

The dynamic call graph in Fig. 2.1 gives an overview of the relationship between each function that was called during both simulations. The majority of the self time in `velocity` is spent solving systems of linear equations that are returned from `assemble`. As will be explained in chapter 4, certain characteristics of an algorithm are better suited for the architecture of a GPU than others. The algorithm in `assemble` is a prime

candidate for a GPU-implementation. A significant portion of the total time spent in `assemble` stems from `greendotleg1` that also includes the time from `greenint`. It is also worth mentioning that the majority of the time in `tprod` originates from calls from `greendotleg1`. The focus for the rest of this thesis will therefore be put on `greendotleg1` and its child functions including the part of assemble from where it is called.

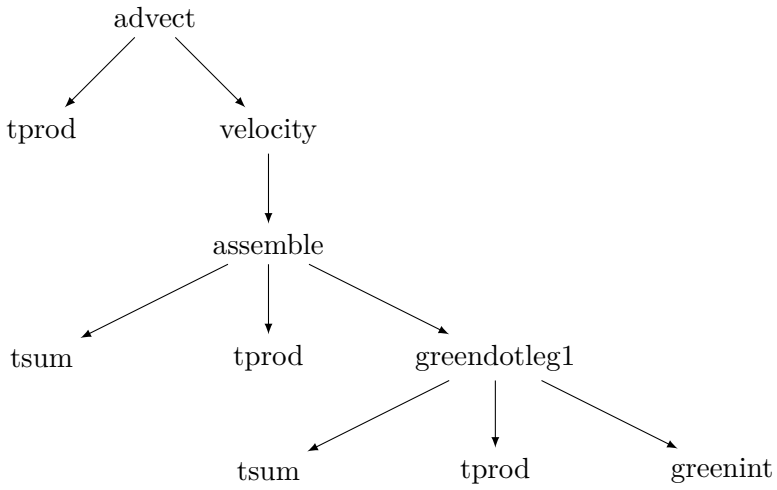


Figure 2.1: Dynamic call graph of the profiled simulations

2.4 Overview of relevant functions

The velocity of a fiber is influenced by all other fibers that participate in the same simulation. The interaction between each fiber pair is computed by the functions `greendotleg1` and `greenint`. In this interaction, the forces are expanded using Legendre polynomials where the number of polynomials can be changed through a parameter to the simulator. As derived in [15], this formula is expressed as

$$\int_{-1}^{1} \left[\int_{-1}^{1} \mathbf{G}(\mathbf{x}_m + s\mathbf{t}_m - (\mathbf{x}_l + s'\mathbf{t}_l)) P_k(s') \, \mathrm{d}s' \right] P_n(s) \, \mathrm{d}s. \tag{2.1}$$

This defines the velocity disturbance due to the interaction of fiber m and l where $\mathbf{G}$ is a Green's function. P_k and P_n is the Legendre polynomial of degree k and n respectively and $0 \leq k \leq N$, $1 \leq n \leq N$ where N is the highest order Legendre polynomial. Here, a fiber parameterized by s is represented by its center coordinates $\mathbf{x}$ and orientation given by its tangent vector $\mathbf{t}$.

2.4.1 `assemble`

The algorithm in `assemble` from where `greendotleg1` is called can be summarized as:

1 Compute all quadratures on each fiber
2 **foreach** *Fiber* **do**
3 | Compute the velocity disturbance for all interactions with the current fiber not previously computed
4 | Partially compute the contribution of the current fiber to the tangential and rotational velocities for the rest of the fibers
5 **end**

Algorithm 2.1: `assemble`

In line 3, `greendotleg1` is called to compute all interactions with the current fiber. This is done once for all fibers. Alg. 2.1 is in turn called once for each time step for the complete duration of the simulation.

2.4.2 `greendotleg1`

The outer integral in Eq. 2.1 is evaluated by `greendotleg1`. The interval of the integration is divided into a number of subintervals that are evaluated using a three point Gauss quadrature rule. The number of subintervals is given as a parameter to the simulator. As each fiber pair interaction only needs to be computed once, given the number of subintervals N_q, the number of interactions that need to be computed in each time step of a simulation with M fibers is $\frac{M(M-1)}{2}N_q$. Each of these interactions can be computed independently which is an important property for a GPU implementation.

2.4.3 `greenint`

In [15], the inner integral in Eq. 2.1 is rewritten and the result is a set of integrals that are defined as

$$L_k^{\alpha,\beta} = \int_{-1}^{1} \frac{s^\alpha P_k(s)}{(\sqrt{s^2+bs+c})^\beta}\,\mathrm{d}s, \tag{2.2}$$

where $b = -2\mathbf{t_l}\cdot\mathbf{R_0}$ and $c = |\mathbf{R_0}|^2$. Furthermore, $\alpha \in \{0,1,2\}$, $\beta \in \{1,3,5\}$ and $0 \le k \le N$.

These integrals are evaluated in `greenint` using analytic three-term recursion formulas where monomials are used to build the Legendre polynomials. Due to some instability in the algorithm, both a forward recursive formula [15] and a backward recursive formula [1] are used. The backward recursive formula is used in the case when a fiber pair is far apart where it produces a more accurate result.

2.4.4 `tprod` and `tsum`

FIBR uses `tsum` and `tprod` for computing different tensor sums and tensor products. These functions can handle general tensor operations between two arrays and are implemented as optimized MEX-functions. They are optimized for the CPU and the internals of these functions are therefore not relevant for the developement of the extension. The complete functionality will not be ported and the focus will instead be on the actual operations performed by their calls from `greendotleg1` and `assemble`.

2.5 Related Work

Solving the N-body problem usually has a time complexity of $O(N^2)$ as is the case with the direct algorithm treated in this thesis. The fast multipole method (FMM) can often reduce this down to $O(N)$ by approximating points that are close together and simplify them into a single point. A GPU implementation of an adaptive FMM using double precision is presented in [2] together with a comparison with a sequential implementation on the CPU using SSE intrinsic instructions. The results showed a speedup of 11 compared to the CPU version. The FMM has a higher constant factor than a direct algorithm and is therefore advantageous only for a sufficiently high number of points. In [2] this number was shown to be above 3500 points when comparing both algorithms running on a GPU.

3 Problem description

The potential gain is greater for a parallelization using a GPU compared to a CPU. A compute intensive part of the FIBR simulator suitable for a GPU was therefore identified in the previous chapter. An extension to FIBR where the identified functions `greendotleg1` and `greenint` are parallelized and ported to the GPU will be designed and implemented. The goal is to improve the performance to allow larger and more realistic simulations to be carried out in a shorter amount of time.

The entry point of the extension will be called from the `assemble` function written in MATLAB. The design will consider the requirements of the GPU and include an interface to the existing code implemented using MEX. The next chapter introduces the GPU requirements before the final design is given in chapter 5.

The implementation will be evaluated with respect to the improvement in speed as well as the accuracy where the existing FIBR functions are used as a reference. The decision of using single-precision or double-precision for the extension is an example of a trade-off that affects both the performance and the accuracy.

3.1 Delimitations

The original codebase will not be optimized and changes will only be introduced here if it is necessary for the functionality of the extension. The focus is on a single GPU so any optimizations that include the CPU or multiple GPUs will not be considered.

4 GPU Programming

When programming GPGPU, several optimizations need to be done manually with the architecture taken into consideration. This chapter will therefore introduce the architecture of the GPU that will be used in this thesis as well as the programming language used for the implementation of the extension to FIBR.

4.1 GPGPU programming language

There are two clear alternatives today when deciding what GPGPU programming language to use, CUDA and OpenCL. CUDA is a proprietary language developed by NVIDIA and is exclusive to their GPUs [9]. OpenCL is developed by the Khronos Group and is an open standard that is supported by a number of GPUs and CPUs from different manufacturers [3].

The GPU that will be used in this thesis is a GeForce GTX 260 from NVIDIA [10]. Both languages were therefore a possible option. CUDA was decided to be the best alternative due to the greater amount of documentation that was available, in particular about using CUDA together with MATLAB. I also had access to people with prior experience with CUDA. The terminology concerning GPGPU can differ depending on the source. This thesis will use the terminology from NVIDIA.

It should be noted that MATLAB has support for executing a subset of its functions on a GPU through its parallel computing toolbox [7]. The implementation of these functions is also made using CUDA. The choice to still make a native CUDA implementation was made due to the following reasons.

- The parallel computing toolbox is not included with MATLAB and would normally require an additional purchase. In this case, it was included in the license provided by my university. This might not be the case for all users and by using CUDA, a GPU can be used without this requirement.
- While an implementation using the parallel computing toolbox would most likely be easier and more straightforward, with native CUDA it is possible to access all features that are available on the GPU. For example shared memory (section 4.5.2) that is not accessible through the toolbox. By performing all operations in the same kernel together with an understanding of the GPU architecture, more optimizations can be made manually and by the compiler with better performance as a result.
- Not all of the functions used by `greendotleg1` were written in MATLAB.

4.2 GPU architecture

The GPU and the CPU evolved with separate goals that resulted in some fundamental differences [4]. The CPU needed to execute programs with very different characteristics with similar efficiency. The GPU was concerned with efficient graphics processing that is a highly data parallel task. As a result, a modern CPU can run a small number of

independent threads simultaneously using various techniques to decrease the latency of a single thread. The GPU on the other hand uses a large number of lightweight threads focusing on the total throughput across all threads.

Limitations in the hardware of the GPU, as a result of the data parallel focus, can cause large performance penalties for general purpose algorithms if they are overlooked. Some of these limitations have been relaxed on newer GPU generations and can therefore differ depending on the GPU model [4, 11, 12].

4.2.1 Geforce GTX 260

The version of the GeForce GTX 260 that was used has 27 streaming multiprocessors (SMs). The threads are distributed to the available SMs and then arranged into groups of 32 threads in each individual SM. One group is called a warp and is associated with the same SM until all threads in that warp have finished their last instruction. An SM can execute the same instruction for an entire warp at a time while each thread in that warp can manipulate different data. This way, the data parallelism in an algorithm can be taken advantage of.

The time it takes to execute a particular instruction for a warp depends on the type of instruction and where the data is located. As the primary goal of the GPU was graphics processing where single-precision is of most importance, every SM has 8 streaming processors (SPs). An SP can execute one single-precision instruction or one integer instruction in one clock cycle [11]. From the currently active warp, 4 of the 32 threads can be assigned to each SP. Executing the same instruction over an entire warp therefore takes 4 clock cycles to complete if the needed data is assumed to have already been loaded into registers. If the instruction is different for some of the threads within the same warp the execution will be serialized for each unique instruction. This is the case when threads take different paths in a branch and can cause a large performance penalty if there are lots of divergent threads.

Double-precision is of little use for graphics processing but required by several GPGPU applications. As the first architecture from NVIDIA to support double-precision operations, each SM also has one double-precision unit. A double-precision instruction will take 32 clock cycles to complete for one warp.

The memory space of the GPU is completely separated from the RAM that is accessible from the CPU. The data that will be processed by the GPU needs to be transferred from the RAM to the memory of the GPU and the results transferred back again to be used by the CPU. As this communication is relatively slow it should be avoided as much as possible. The memory space of the GPU is divided into a larger global memory that is located off-chip and smaller but faster on-chip registers and shared memory inside each SM.

The way the memory is used has a large impact on the final performance. Reading and writing to the global memory has a latency of 400-600 clock cycles [12]. Two caches to the global memory are available, texture memory and constant memory. There is however no automatic caching so frequently used data needs to be explicitly moved to faster memory. This can either be done during run-time by loading data from global

memory to registers and shared memory, or before launch by assigning a portion of the data to be cached. The memory accesses that are made by the threads in the same warp can in some cases be performed more efficiently if consecutive memory addresses are accessed and are therefore preferred to scattered memory accesses. The different types of memory and the access patterns they are optimized for will be explained in more detail in section 4.5.

To cope with the various latencies that the memory accesses introduce, a warp that is stalled due to a memory access can be replaced by another warp. This can be done without any overhead using a hardware scheduler in each SM and by reserving the registers and shared memory for all currently active warps [11]. The occupancy of an SM is measured by the number of warps that can be active at the same time. This should be high enough to hide the latency and depends on the number of threads launched and the amount of resources required by each thread.

4.3 Desirable characteristics of an algorithm

Using this brief overview of the architecture, a suitable algorithm for the GPU can be defined:

- The algorithm should be highly data parallel, the same computations on a large number of different elements that can be partitioned into a large number of threads.
- The computational complexity on each element should be high enough to compensate for the overhead of memory transfers between the RAM and GPU.
- The number of divergent branches should be small, or separable between different warps.
- The majority of the memory accesses should take advantage of spatial locality for better efficiency.

4.4 CUDA

The CUDA programming language uses a subset of the C programming language with additional extensions for the functionality of the GPU [11]. A program written for the GPU is called a kernel. As the number of SMs and SPs can differ between GPUs, a kernel needs to be scalable. CUDA deal with this by using an abstraction of the architecture where the kernel is divided into a grid of thread blocks, where each block is a collection of warps. Blocks are assigned to an SM during the execution of the kernel in an order that is undefined [11]. Efficient thread communication, using shared memory, is therefore only supported for threads within the same thread block since they are guaranteed to reside on the same SM where the shared memory is located.

All threads run the same kernel. To perform different actions, the threads use their unique id together with the id of the block they belong to. This is typically used as a way to indicate what data element each thread should work on but can also be used to

decide what branch to take. The id for a thread is equal to its coordinate in the block, and the block id is equal to the coordinate in the grid. Both the grid and the blocks can have up to three dimensions.

This abstraction influences how an algorithm is adapted to the GPU. Parts of an algorithm where the same data is reused multiple times can be handled by threads in the same block that share common data. This removes unnecessary accesses to global memory. The independent parts of the algorithm can then be divided into separate blocks. The dimensions and their size for the grid and the blocks are set in a way that fit the algorithm.

There can be more differences than the number of SMs and SPs. This is reflected using a number, called compute capability, to indicate a particular set of functionality for a GPU. The GeForce GTX 260 has compute capability 1.3. The specific details of this number are not relevant at the moment, but the details of what will be explained in the remainder of this chapter apply to this particular compute capability.

4.4.1 Limitations

- A kernel can only call functions that are written for the GPU. Functions from the C standard library for example, are not available. Various mathematical functions callable from a kernel are however provided by CUDA.
- No dynamic memory allocation during the execution of a kernel. Memory needs to be allocated in advance before the kernel is invoked.
- There are no function pointers. Recursion is therefore not possible and function calls will be inlined at compilation.

4.5 Different memory types

This section will outline the different memory types that are available on the GPU and the access patterns that they are optimized for. This is a condensed summary of [11] and [12] that explain the differences in more detail. The exact size of each memory type is listed in appendix A.2.

4.5.1 Register memory

Registers are the fastest type of memory and are private to each thread. The number of registers required by each thread is called the register pressure. There is no particular usage pattern that needs to be considered, only that the register pressure affects the number of threads that can be active at a time on each SM since the available registers are distributed between all active threads.

4.5.2 Shared memory

The shared memory is shared between all threads in the same block. Data that is written to shared memory by a thread can be read by the other threads in the same block after

a synchronization.

The memory space is divided into 16 memory banks and the elements of an array are stored into consecutive memory banks. Memory accesses are split into a separate memory request per half-warp. This makes the shared memory very bandwidth-efficient since all 16 threads in a half-warp can read and write from the array simultaneously if they access elements from separate memory banks. All accesses to different elements that are located in the same memory bank will be serialized.

4.5.3 Global memory

The data that will be processed is transferred to global memory before a kernel launch. This is the only memory that is accessible from the CPU and persistent between different kernel launches. The final result of the computations should therefore be stored in this memory as well as intermediate results that are too large to fit the smaller but faster memory types.

To minimize the latency, memory accesses can be coalesced. The memory space is divided into segments and if all accesses by a half-warp are made to the same segment they can be serviced by one transaction. The segment sizes are 32-, 64- and 128 bytes and the smallest segment size that is possible is used in each transaction.

4.5.4 Local memory

If threads require more register memory than is available, the data is spilled into local memory instead. Local memory is actually the same type of memory as global memory but is private for each thread. It should be avoided because of the high latency.

4.5.5 Constant memory

This is a read-only cache for global memory. Data needs to be assigned to constant memory before the launch of a kernel. It is optimized for broadcast of the same element where it can be as fast as a register [12]. A separate memory request is made per half-warp and if threads within the same half-warp access different elements, each access will be serialized.

4.5.6 Texture memory

Texture memory is also a read-only cache for global memory. As with constant memory, data needs to be assigned to texture memory before the kernel is launched. In [12], this memory is recommended where the access pattern does not fit the previously mentioned memory types but takes advantage of 2D spatial locality. It is also possible to use texture memory to automatically interpolate data. This memory type was not used in the extension as the other memory types were better suited.

5 GPU implementation

The extension is composed of two parts, the entry point and interface from MATLAB in the form of a MEX-function and the second part that is the kernel itself. The structure of the kernel was kept as similar as possible to the original implementation, with `greendotleg1` and `greenint` as separate functions in the kernel. The switch between the GPU-implementation and the original and unchanged code is made using an additional parameter that was added to the simulator.

Due to the lack of prior experience with GPGPU and CUDA, and also with FIBR, the first step in the development of the kernel was to make a rough translation of the MATLAB functions `greendotleg1` and `greenint` to the C programming language. The high-level nature of the syntax in MATLAB and its functions can make it difficult to get an overview of the underlying operations and the required memory accesses. The result of the translation could be ported to CUDA in a straightforward way after the level of parallelization had been decided.

5.1 Tools

The profiler that is included in the MATLAB IDE does not support CUDA. To profile the kernel a basic command line profiler that is included with CUDA was used. This profiler provides a way to gather some useful statistics during the execution of a kernel. To provide this information the profiler uses some hardware counters that are available on the GPU. The compute capability 1.3 has a limit of 4 counters so multiple profiles were necessary when more statistics were required. The final result is stored in a text file after the completion of the kernel. The statistics that were used were:

- The number of unique branch instructions executed by the kernel and the number of these instructions that diverge within a warp.
- The number of instructions executed by the kernel.
- The number of warps that serialize because of a non-optimal access pattern to shared memory and constant memory.
- The number of coalesced global memory loads as well as the number of non-coalesced loads due to a non-optimal access pattern.
- The number of coalesced global memory stores as well as the number of non-coalesced stores due to a non-optimal access pattern.

For debugging of the kernel cuda-gdb was used. This is an extended version of the GNU Project debugger that makes it possible to step through the execution of a kernel and examine variables. This is very useful since the only alternative was to transfer all values to be examined from the GPU back to the CPU which is not a very flexible solution. Using any print functions from the GPU is not supported.

5.2 Parallelization of `greendotleg1` and `greenint`

The interactions that would be computed for 7 fibers in `assemble` using `greendotleg1` and `greenint` can be visualized as shown in Fig. 5.1. The current fiber is given by the x-axis and the fibers it interacts with are given by the y-axis. The loop in `assemble` iterates over the x-axis one step at a time from left to right indicated by the arrow. Interactions that are not computed are marked with an "X". This includes the interaction with the current fiber itself and previously computed interactions.

The number of subintervals that are used when evaluating the outer integral of Eq. 2.1 increases the number of interactions that need to be computed. In Fig. 5.2, two subintervals have been introduced to the previous setup resulting in two quadratures to evaluate in each fiber pair interaction. The evaluation of one quadrature is the smallest computation where the same sequence of operations is performed on the different input data. The GPU is optimized for a large number of threads so in this kernel each thread is assigned to one quadrature. This creates a large number of threads that can be partitioned into enough blocks with even workload to keep all SMs busy.

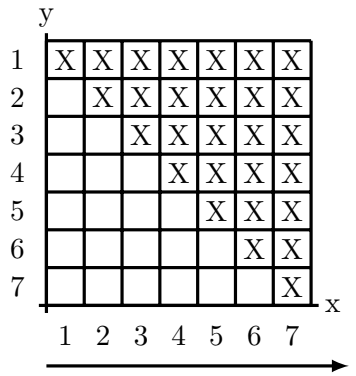


Figure 5.1: Fiber-to-fiber interactions

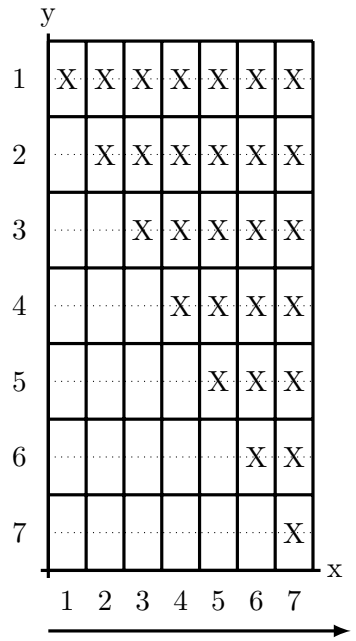
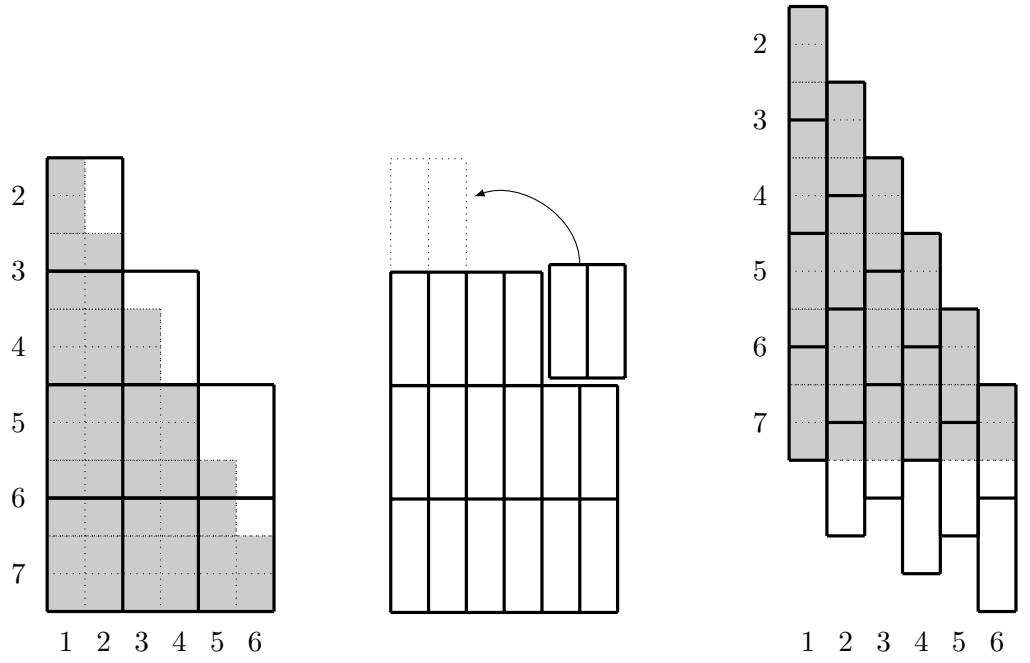


Figure 5.2: Fiber-to-fiber interactions with two quadratures

Using this parallelization as opposed to assigning more work to each thread, several loops can be avoided. They are instead implicitly expressed using the id of the threads to indicate what data element to work on. As the same code is executed in all threads but with a different id, the whole range of the input can be processed by launching enough threads. This decreases the complexity of the kernel and can make it easier to control the register pressure.

5.3 Grid and block configuration

The grid of thread blocks that the kernel is launched with is not perfectly suited to the triangular shape of the interactions so some blocks might need to be rearranged after the kernel has been launched. The threads are partitioned into one-dimensional blocks that are arranged into a two-dimensional grid. The blocks are distributed so that they cover all interactions and can be rearranged from the two-dimensional grid using their coordinates. For the purpose of illustration, we assume that the block size is 3 threads. The number of blocks required for the interactions in Fig. 5.2 is calculated by distributing them in a triangular-like fashion shown in Fig. 5.3a. These blocks are then launched in a grid where two blocks are reassigned to their position after the kernel has been launched (Fig. 5.3b). Finally in Fig 5.3c, each column is adjusted so that the first threads in a block with interactions are always occupied and the threads without work are at the end of the block. This ensures that as many warps as possible are filled with work so that the number of memory transactions can be minimized due to coalescing.



(a) Needed blocks to cover all interactions

(b) Blocks are launched in a grid and then rearranged

(c) Final adjustments of the block positions

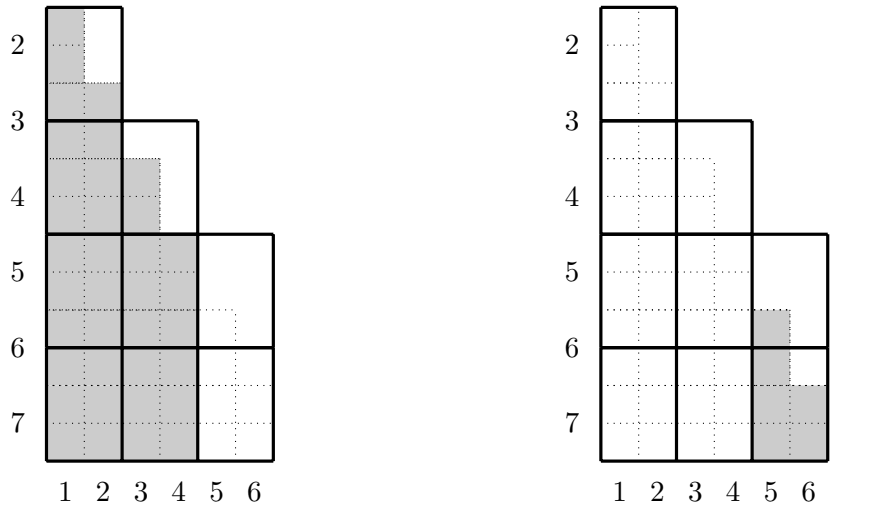
Figure 5.3: Distribution of thread blocks

The real block size should be a multiple of the warp size (32 threads) because of the considerations mentioned in section 4.2.1 and 4.5. By distributing the blocks in this way there are no restrictions on the number of quadratures in each fiber pair interaction that depend on the block size. There will however be some extra blocks launched and some underutilized blocks if the block size is not a multiple of the number of quadratures.

5.4 Amount of work in each kernel invocation

The loop in `assemble` requires that the result is presented in the order shown in Fig. 5.1 and Fig. 5.2, from left to right on the x-axis. Each invocation of a kernel will introduce some overhead but in this case there is no reason to invoke the kernel for every iteration of the loop. Instead, as much as possible will be computed and returned to the CPU in each invocation.

The amount of global memory available on the GPU will vary between different models and since the simulations can be large, the interactions can not always be computed all at once. The MEX-function will split the computations into chunks that fit the available GPU-memory before invoking the kernel. Each chunk will contain the required results for the current iteration and as much of the future iterations as possible. The MEX-function will only be called again when `assemble` has processed the entire chunk. In Fig. 5.4a only the first four iterations fit the memory of the GPU. In the fifth iteration the MEX-function will be invoked again and the next chunk of results will be returned (Fig. 5.4b).



(a) The first chunk

(b) The second and last chunk

Figure 5.4: The interactions are split into chunks that fit the memory of the GPU

An optimization is made here where the next kernel call is launched immediately after the results of the previous call have been transferred from the GPU to the RAM. As the CPU and the GPU are separate devices, they can both be used at the same time. With CUDA each kernel call is asynchronous and will return immediately, before the kernel has finished. A synchronous memory copy will be used the next time the MEX-function is invoked to transfer this chunk of results from the GPU. The CPU will therefore use the results from the previous chunk for further processing at the same time as the GPU is computing the next chunk.

5.5 MEX-function

The MEX-function that is called from the `assemble`-function will take care of all communication with the GPU including each invocation of the kernel. It will also perform the initialization that is needed before the launch of the kernel and keep track of the state of the execution. The functionality of the MEX-function can be summarized as:

```
1  if not initialized then
2  |  Allocate global memory on the GPU
3  |  Copy constant data to the GPU
4  end
5  if no previously computed results then
6  |  Check how much that can be computed on the GPU
7  |  Set block and grid parameters
8  |  Copy variables used for the triangular rearrangement of blocks to the GPU
9  |  Asynchronous launch of kernel
10 end
11 Synchronous copy of results from GPU to RAM
12 Precompute the next chunk (by repeating line 6-9)
13 Prepare the results to be returned to assemble
14 if last chunk is about to be returned then
15 |  Clean up and deallocate GPU memory
16 end
```

Algorithm 5.1: MEX-function, the entry point

5.5.1 Initialization

The initialization in line 1-4 is made once in each time step of the simulation or in rare cases when the mex-file has been prematurely cleared from memory by MATLAB. To avoid any redundant transfers between the RAM and GPU, all constant data will be copied to the GPU here. This includes the center coordinates of all fibers and their orientation. Any global memory that will be used by the blocks during the execution of the kernel will also be allocated here due to the lack of dynamic allocation on the GPU.

5.5.2 Termination

The MEX-function has no notion of the number of time steps in the simulation and since the majority of the data transferred here to the GPU will change in each time step, the MEX-function will clean up after each completed time step (line 14-16). This portion of the code is also registered to MATLAB as the exit function and will therefore be called if the MEX-function needs to be cleared from memory prematurely.

5.6 Accuracy of the kernel

While the original implementation of FIBR uses double-precision, the initial version of the kernel was implemented using single-precision due to the fact that the GPU that was used was more optimized for single-precision. The final error was however larger than anticipated. With the unmodified FIBR as a reference, the progression of the error for 5 fibers from 0 to 800 time units and a time step of 0.04 is shown in Fig. 5.5. The graph depicts the mean value of the relative error in each time step. To examine if this was

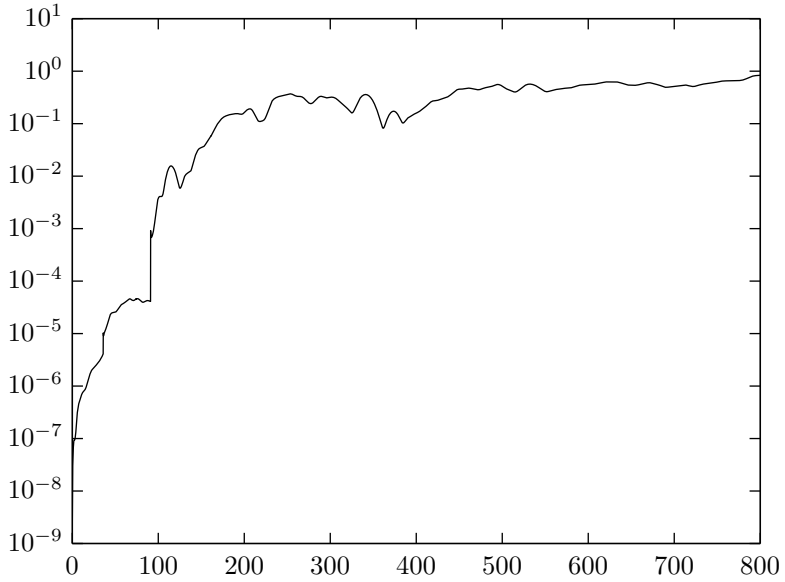


Figure 5.5: Relative error using the single-precision kernel

caused by the use of single-precision or an error in the kernel, another set of results was produced using the unmodified FIBR but with single-precision perturbation of the input. In Fig. 5.6 the relative error of this result, represented by the dotted line, was computed and added to the previous figure. Here we can see that the unmodified FIBR has the same behaviour as the kernel, indicating that the error is caused by the sensitivity of the simulator to round-off errors.

In [15], the recursive formula that is used in the evaluation of the inner integral in Eq. 2.1 is pointed out to have this sensitivity. This could also be observed when manually examining this part of the computation with the debugger. The use of single-precision in the kernel is therefore not suitable in this scenario and the kernel was modified to include support for double-precision. The precision to use is changeable at compile-time with double-precision as the default choice. The support for single-precision was kept since it is useful to still be able to run the kernel on GPUs without double-precision support.

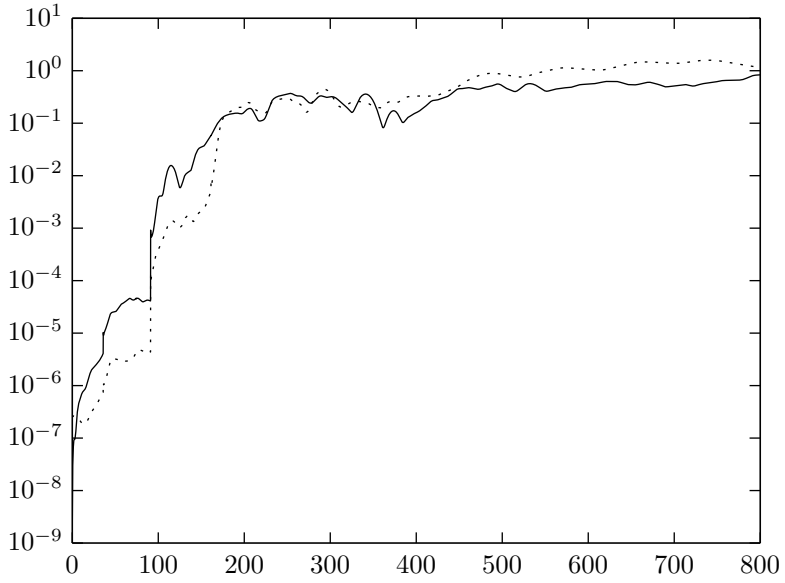


Figure 5.6: Relative error with perturbed input, CPU-only (dotted line)

5.7 Kernel optimizations

The optimizations that were made to the kernel were primarily focused on the compute capability 1.3. More modern generations of GPUs are often more lenient but there are some exceptions. Compute capability 2.x has a maximum limit of 63 registers per thread compared to 128 registers with compute capability 1.3. The goal for the maximum register usage was therefore set to the 2.x limit to avoid register spilling to local memory on that architecture as well.

One register can store one 32-bit value so a double-precision value will require two registers. The switch from single-precision to double-precision therefore nearly doubled the required registers per thread. Whether a register should be used or not is decided by the compiler that will automatically try to optimize the kernel by using registers to avoid repeated global memory accesses and calculations. These optimizations can be a bit limited and manual optimizations are therefore often required.

The register pressure of the kernel was initially higher than desirable and limited the number of warps that could be active per SM. This could be partially fixed by simply removing repeated computations and memory loads by using temporary variables as much as possible.

5.7.1 Reducing the instruction count

The available alternatives to reduce the instruction count were limited. CUDA includes some intrinsic mathematical functions that require less instructions but with less precision. These were not considered because of the importance of the accuracy of the

results.

The modulo operator was used in some parts of the code that were called repeatedly. These occurrences were of the type "n mod 2" that can be replaced with the equivalent "n & 1" where the "&" is the bitwise AND operator that is more efficient. This was however overlooked by the compiler and was therefore done manually.

An improvement that required an interesting circumvention is shown below where the original version in Alg. 5.2 was changed to Alg. 5.3. The intent of the improved version is to store the values in an array using registers instead of recomputing the values at every iteration. Because of a limitation where the registers are not dynamically addressable, the compiler will put d in local memory instead of registers. This was fixed by rewriting line 4 in Alg. 5.3 as the conditional assignment $r = (i \bmod 2)\ ?\ d[0] : d[1]$.

```
1 b = −1;
2 foreach (Iteration i) do
3 |   ...
4 |   r = a − b ∗ c;
5 |   ...
6 |   b = −b;
7 end
```

Algorithm 5.2:

```
1 d[2] = {a − c, a + c};
2 foreach (Iteration i) do
3 |   ...
4 |   r = d[i mod 2];
5 |   ...
6 end
7
```

Algorithm 5.3:

5.7.2 Common operations and data

Constant data where all threads read the same element were put in constant memory where it can be cached and broadcast to threads in the same warp. Common data computed during the run-time of the kernel and data that was to large to be put in constant memory were shared within each thread block using two synchronization points at the beginning of the kernel. Here only the required data for each thread block is loaded into shared memory by a subset of the threads.

The first synchronization point involves the calculation of the thread block position that decides what interaction each thread will be assigned to. Every thread will use their id as the offset from this position. This computation is performed by the first thread in each thread block. As discussed in section 5.3 some superfluous blocks will be launched and threads in any block where the position falls outside the boundaries will terminate after this synchronization.

In the second synchronization point, common data such as the center coordinate and tangent vector for the fiber that is associated with each thread block are shared. The memory address for the portion of the global memory that belongs to the thread block is also calculated and shared here.

5.7.3 Memory layout for intermediate results

The intermediate results that are computed by the CUDA version of `greenint` were to large to be stored in registers and shared memory and are therefore stored in global memory. These results are returned to `greendotleg1` for further processing before the final results can be transferred to the RAM.

As every memory request to global memory is made per half-warp, 16 threads, and the size of a double is 8-bytes the total size of the request when all threads are active is 128-bytes. This is the size of the largest transaction that is available with compute capability 1.3. If every thread is arranged to access consecutive elements they can be coalesced into one transaction per memory request but only if all accesses are aligned with the 128-byte segments in global memory.

When the thread blocks are positioned, some threads will fall outside the boundaries and will not be assigned any interactions. Since the global memory is allocated as one single large memory block for all thread blocks, a naive way of allocating global memory can create a misalignment that will propagate to other thread blocks as shown in Fig. 5.7. This requires multiple transactions per memory request. To get the optimal access pattern, the global memory was allocated also for useless threads so that the size of the memory block that is assigned to all thread blocks is kept as a multiple of the warp size. This creates perfectly coalesced accesses for all intermediate results and is therefore more important than the extra space required (Fig. 5.8).

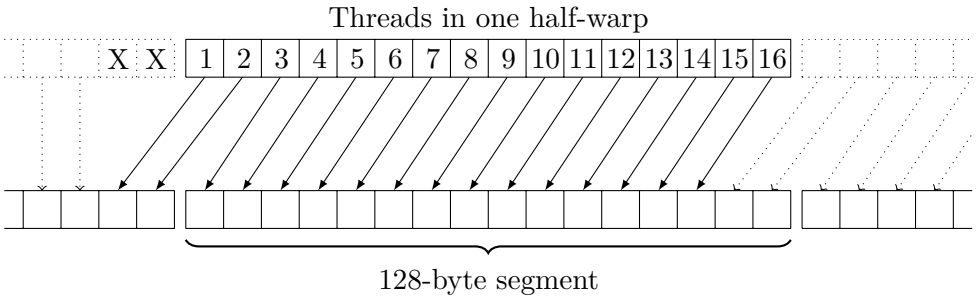


Figure 5.7: Non-optimal access pattern to global memory

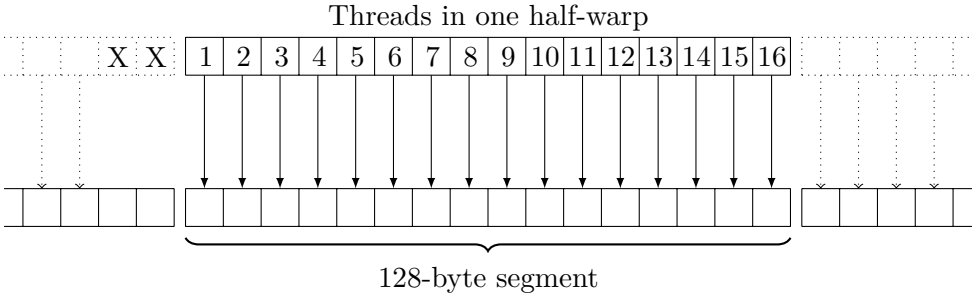


Figure 5.8: Optimal access pattern to global memory

5.7.4 Buffered results in the recursion formulas

The two three-term recursion formulas in `greenint` are both implemented using a loop that generates and stores data in every iteration using some of the previously generated values. This caused multiple redundant accesses to global memory where the data is stored. A shared memory buffer was implemented to temporarily store these values as long as they will be reused while still storing the generated values in global memory. This reduced both the register pressure and the number of accesses to global memory.

Every thread requires a buffer space of six elements and since they work on separate data there is no synchronization needed before accessing the buffer. The optimal access pattern to shared memory when all threads work with different data is to let each individual thread access separate memory banks. With single-precision this is accomplished by simply treating the shared memory buffer as six rows, where one row has the size of one thread block, and then letting every thread access consecutive elements in the buffer as shown in Fig. 5.9.

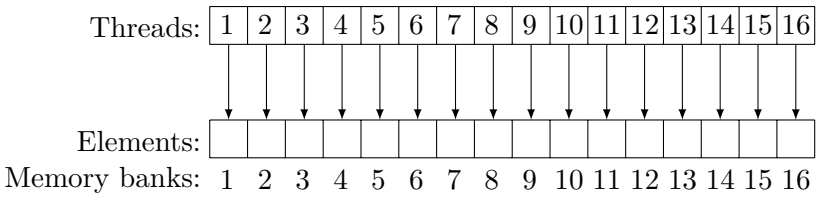


Figure 5.9: Optimal access pattern to shared memory

With the change to double-precision, the profiler reported a large number of bank conflicts using the above pattern. The reason for this was that only 16 memory banks are available and one double element will occupy two memory banks. When a memory request is made to a double, two separate 32-bit accesses are made automatically. The first access when a half-warp request consecutive elements is shown in Fig. 5.10. Here, the lower half of the threads access the same memory banks as the upper half resulting in serialized accesses for all conflicting threads.

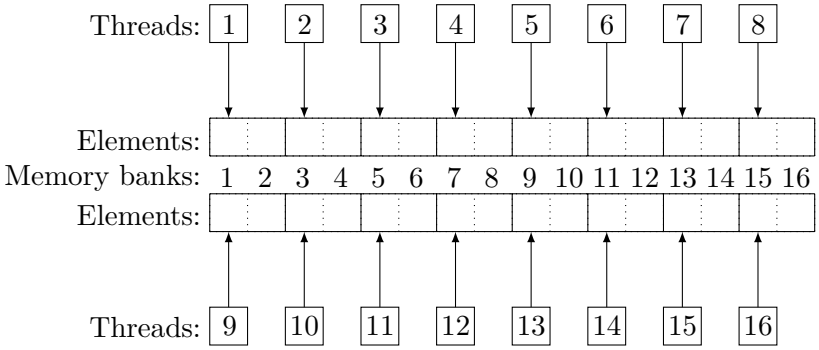


Figure 5.10: Non-optimal access pattern to shared memory

To avoid this the upper and lower bits of all double-precision elements were split into two separate buffers using some intrinsic functions available in CUDA. Two accesses were still required but without any bank conflicts (Fig. 5.9). This did actually perform slightly worse and while the profiler reported that the bank conflicts were removed, the number of instructions in the kernel had increased. This is not surprising considering the extra steps required to split the doubles to be stored into the buffer and putting them together again when loading them from the buffer. As the number of instructions was more important to the performance than the bank conflicts, the original implementation with one buffer was used. Accessing consecutive double-precision elements in shared memory in this way will only introduce bank conflicts with compute capability 1.3 and is handled better in newer architectures with for example compute capability 2.x [11].

5.7.5 Building the Legendre polynomials

The next step in `greenint` after the recursion formulas have finished is to build the Legendre polynomials. This is done by repeated calls to a subroutine that is given in Alg. 5.4. The subroutine is fairly straightforward, N is the number of Legendre polynomials + 1, d_matrix contains a part of the results from the recursion formulas for every thread (N rows with the size of one thread block) and s_M2L is an N by N monomial to the Legendre transform matrix.

```
1 for (i = 0; i < N; i++) do
2 |   sum = 0;
3 |   for (j = 0; j < N; j++) do
4 |   |   sum += d_matrix[j * BLOCK_SIZE + threadId] *
5 |   |           s_M2L[i * N + j];
6 |   end
7 |   d_result[i * BLOCK_SIZE + threadId] = sum;
8 end
```

Algorithm 5.4: Function to build the Legendre polynomials

With regard to memory optimizations there are several improvements that can be made here. In the algorithm given, s_M2L has already been placed in shared memory. This gave a noticeable improvement to the performance. Since all threads access the same element it can be broadcast to all threads in the same half-warp. This pattern is also ideal for the constant memory, but shared memory proved to perform better.

Another source of improvement is the global memory loads from d_matrix in the inner loop, where the same elements are reloaded N times because of the outer loop. By using the previously defined shared memory buffer we can instead store six partial sums at a time per thread if we use a tiled version of the algorithm. The tiled version is given in Alg. 5.5. As was shown by the profiler, this version made a significant reduction of the total global memory accesses in the kernel but required a large portion of additional instructions and performed worse. Alg. 5.4 was therefore the final version used in the kernel.

```
1  for (i = 0; i < N; i = i + 6) do
2  |  limit = min (6, N − i);
   |  // Initialize buffer
3  |  for (j = 0; j < limit; j++) do
4  |  |  s_buffer[j * BLOCK_SIZE + threadId] = 0;
5  |  end
   |  // Calculate sums
6  |  for (k = 0; k < N; k++) do
7  |  |  mat = d_matrix[k * BLOCK_SIZE + threadId];
8  |  |  for (l = 0; l < limit; l++) do
9  |  |  |  index = l * BLOCK_SIZE + threadId;
10 |  |  |  sum = s_buffer[index];
11 |  |  |  sum += mat * s_M2L[(i + l) * N + k];
12 |  |  |  s_buffer[index] = sum;
13 |  |  end
14 |  end
   |  // Write results to global memory
15 |  for (m = 0; m < limit; m++) do
16 |  |  sum = s_buffer[m * BLOCK_SIZE + threadId];
17 |  |  d_result[(i + m) * BLOCK_SIZE + threadId] = sum;
18 |  end
19 end
```

Algorithm 5.5: Tiled version of Alg. 5.4

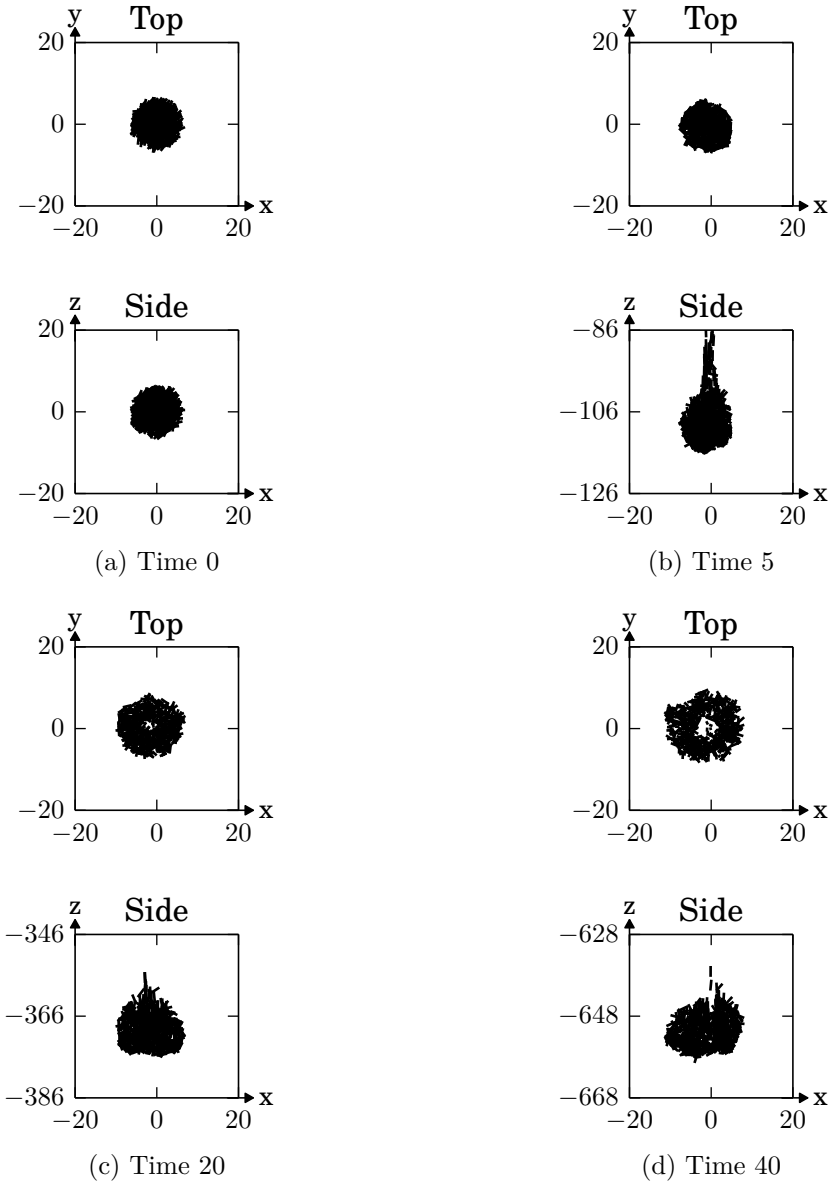
Since the kernel did not respond well to the memory optimizations that increased the instruction count in this and the previous section, it was clear that the kernel at this point was compute bound instead of memory bound. No obvious way of optimizing the memory without additional instruction overhead was left so no further optimizations directly related to the memory were made.

5.7.6 Thread block size and occupancy

The final register pressure of the kernel ended at 61 registers per thread when compiling to compute capability 1.3 and fewer registers when compiling to a higher compute capability. The optimal size of the thread blocks considering the required registers and shared memory of the kernel could be calculated with the CUDA Occupancy Calculator spreadsheet included with CUDA. The size that maximized the possible occupancy for this kernel was 128 threads. With this size two thread blocks can be active at a time per SM leaving 8 warps available to the scheduler.

6 Results

To evaluate the performance, the run times were measured and compared in multiple simulations where the fibers were randomly distributed inside a sphere with a radius of 6 units and then sediment due to gravity from time 0 to time 40. The step size was set to 0.5 time units creating 81 time steps to simulate. An example of the result from a simulation with 700 fibers is shown in Fig. 6.1. This shows the clustering of the fibers



(a) Time 0

(b) Time 5

(c) Time 20

(d) Time 40

Figure 6.1: A simulation with 700 fibers

into a ring shaped formation that is formed when a group of fibers is initially placed inside a sphere. In this example the simulation ends before the fibers disperse. For clarity, all fibers that are out of scope in the side view are also hidden in the top view.

The workload in the simulations was gradually increased by 100 fibers from 100 fibers initially up to 1000 fibers. The rest of the parameters to the simulator were kept at their default value. These include five terms in the Legendre force expansion and 24 quadratures. The first call made to CUDA after starting the MATLAB-environment will have an additional overhead of up to several seconds due to the creation of the CUDA context. This will normally only happen once until MATLAB is restarted. A dummy simulation using the kernel was therefore made before the timed simulations to avoid skewing the results. The elapsed time was recorded for both the portion of the extension alone and for the whole simulation. These values can be found in appendix B and the specifications of the system in appendix A. Since MATLAB uses multiple threads internally for some supported functions and since the extension uses the GPU, the measured times were simply compared directly with each other.

6.1 Performance of the kernel

The kernel was evaluated by timing all calls to the MEX-function from `assemble` until they completed. The parallel execution of the GPU and CPU was temporarily turned off to get an accurate result for the whole execution of the MEX-function and the kernel. The equivalent portion on the CPU was measured by timing all calls to `greendotleg1` and therefore implicitly also `greenint`. The total time in minutes for this part of the simulation is shown in Fig. 6.2.

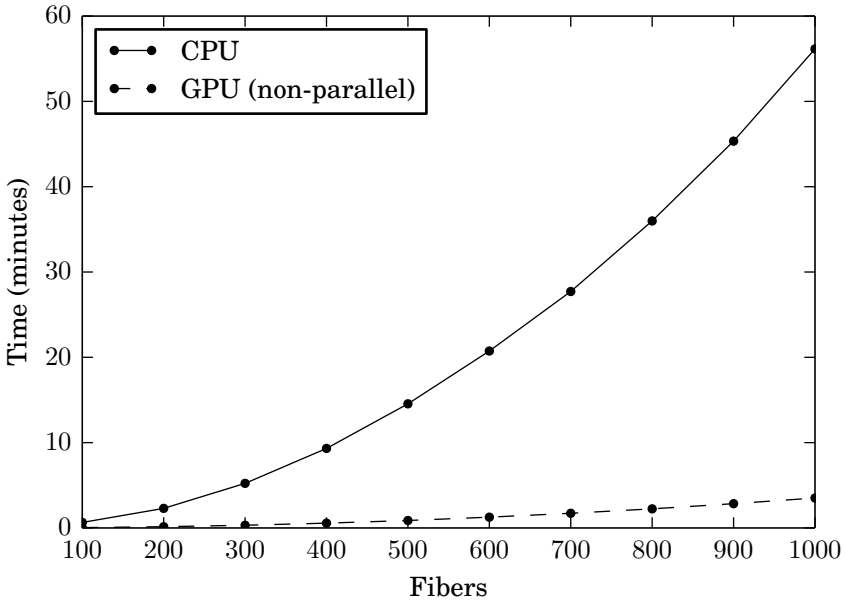


Figure 6.2: Elapsed time for `greendotleg1` and `greenint`

The CPU to GPU time ratio from these results is computed in Table 6.1. This gives an average ratio of 16.27 and it is therefore safe to say that the extension is 16 times faster than the CPU implementation using the default parameters.

Fibers:	100	200	300	400	500	600	700	800	900	1000
Ratio:	16.84	15.72	16.50	16.54	16.64	16.40	16.11	16.07	15.94	16.00

Table 6.1: CPU to GPU time ratio

6.1.1 Parallel execution of the kernel

The elapsed time using the CPU and GPU in parallel is given in Fig. 6.3 together with the non-parallel GPU time. The difference between the two should roughly correspond to the computational time taken by the GPU; and the measured parallel time, the dashed line, to the total data transfer time between the GPU and CPU. The portion of time that can be hidden by the parallel execution increases with the number of fibers and with 1000 fibers this corresponds to more than half of the total time of the kernel.

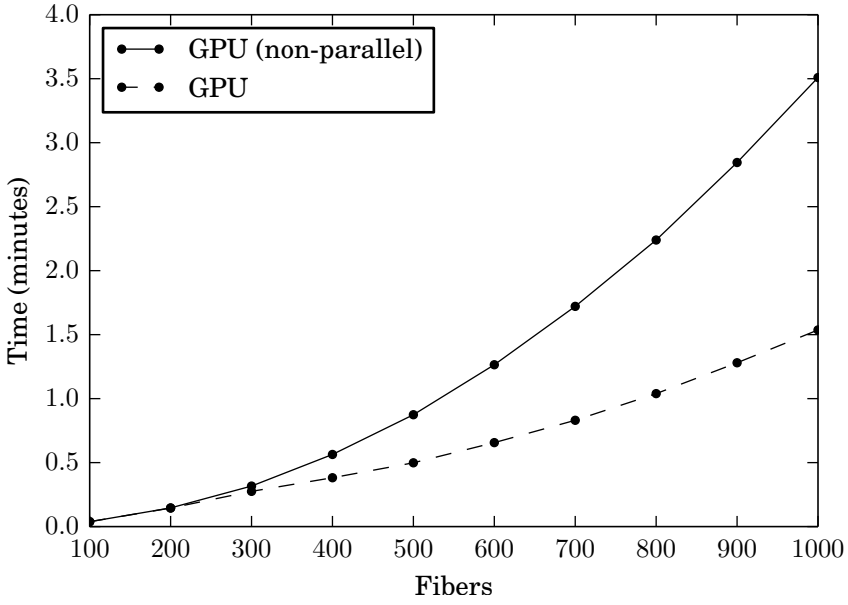


Figure 6.3: Elapsed time for `greendotleg1` and `greenint` (GPU-only)

6.2 Final improvement using the extension

Fig. 6.4 shows the total elapsed time for each complete simulation with and without the extension. Here the parallel execution of the CPU and GPU is enabled again, as is the normal behaviour of the extension. With 1000 fibers a simulation using the CPU takes roughly 3 hours to complete. The extension gives an improvement close to 53 minutes with the same number of fibers.

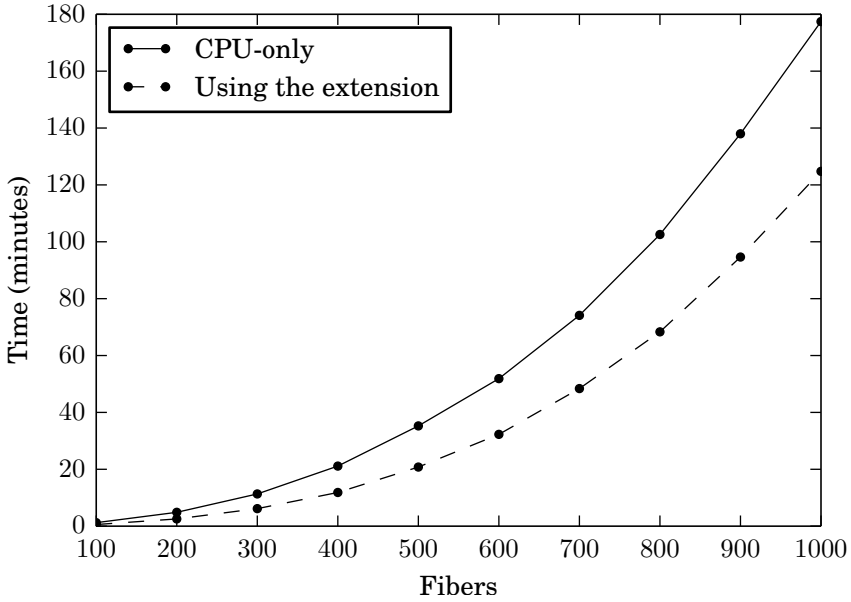


Figure 6.4: Elapsed time for the entire simulation

The percentage improvement of the total time in each simulation using the extension is given in Table 6.2. With an increasing number of fibers the total gain decreases. To be able to interpret these results the portion of the total time that `greendotleg1` and `greenint` is responsible for, using the original implementation, was also computed in Table 6.3. As the number of fibers increases, this portion decreases.

Fibers:	100	200	300	400	500	600	700	800	900	1000
Gain:	53%	48%	46%	44%	41%	38%	35%	33%	31%	30%

Table 6.2: Improvement of the total time of a simulation using the extension

Fibers:	100	200	300	400	500	600	700	800	900	1000
Portion:	52%	47%	46%	44%	41%	40%	37%	35%	33%	32%

Table 6.3: Portion of total time spent in `greendotleg1` and `greenint` (CPU-only)

7 Conclusions

7.1 Discussion

There are multiple parameters that can be changed in the FIBR simulator that will affect the workload on different parts of the simulator in various ways. To produce the results in a way that was as fair as possible without testing all different combinations, the default value on these parameters was simply used. The exception to this being the number of fibers to control the workload and the number of time steps to simulate. The number of time steps was set large enough to produce the ring formation previously shown in Fig. 6.1 and to produce a moderately large simulation.

Given this configuration the results showed that the GPU was about 16 times faster than the original implementation of `greendotleg1` and `greenint` regardless of the number of fibers that were used in the simulation. The results also showed that when using the extension with an increasing number of fibers the total gain in the whole simulation decreased. As the kernel did not show any tendency to perform worse with a larger amount of fibers, the explanation to this behaviour was instead searched for by examining how the influence of `greendotleg1` and `greenint` to the total time of a simulation changed in relation to the number of fibers. These numbers presented in Table 6.2 and Table 6.3 did show a decrease consistent with each other explaining the decreasing gain with an increasing computational workload on other parts of the simulator.

7.2 Conclusion

The goal of this thesis was to improve the run-time of the FIBR simulator by porting `greendotleg1` and `greenint` to the GPU. This was accomplished by investigating the requirements of the GPU used during the development and then rewriting the given functions using CUDA. An interface to MATLAB was implemented using MEX to be able to call the kernel from MATLAB. The kernel proved to be 16 times faster than the original implementation of the ported functions and was able to improve the run-time in all simulations that were performed by 30% to 53%.

A problem with the precision of the kernel was initially discovered during this process showing that the use of single-precision was unsuitable. Fortunately, the GPU that was used during the development had support for double-precision and the kernel was modified to take advantage of this. As long as the kernel is compiled for double-precision this is no longer a problem.

7.3 Future work

7.3.1 Periodic boundary conditions

As briefly mentioned in section 2.1, the FIBR simulator can also replicate the fibers using periodic boundary conditions in addition to the option of considering them in free space as has been done in this thesis. This uses some additional functions in a separate branch in `assemble` apart from free space. The functions ported in this thesis are also used but

with a variation in the input. Here the closest periodic image of the fibers is used when computing the interactions for each fiber. The extension could be used here with some modifications to the kernel to handle this.

7.3.2 Additional work in `assemble`

The rest of the `assemble`-function is also heavily data parallel and could probably be successfully ported to the GPU. This would require another block and grid configuration and because of the complexity of the current kernel, high register usage and other occupied resources, another separate kernel should be implemented. The results that are computed by the extension in this thesis could be stored on the GPU without being transferred to the RAM and then be further processed directly by launching this new kernel.

7.3.3 Improved memory management by the kernel

Further improvements to the kernel performance will not make much of a difference to the total time of a simulation. If more of the `assemble`-function is ported to the GPU in the future it could however be desirable to increase the amount of interactions that can be computed at a time per kernel invocation. The interactions are currently split into chunks of different sizes depending on the memory usage of the kernel and the available global memory on the GPU (section 5.4). Reducing the global memory requirements of the kernel will increase the size of these chunks.

The current implementation allocates all memory once before the kernel is invoked. Separate memory is allocated for all thread blocks even for temporary memory due to the lack of dynamic memory allocation on the GPU used. Manual memory handling of this temporary memory might be possible to implement by querying the number of SMs and the maximum number of threads per SM from CUDA. This can be used to compute an upper bound on the number of thread blocks that will be active at a time. The memory could be allocated only for this number of blocks and then managed at runtime in two synchronization points in the kernel using atomic operations to communicate occupied and released memory to other blocks. This was not tested and could prove to be inefficient. Another alternative is to drop the support for compute capability 1.3 and lower and use the native dynamic memory allocation support.

References

[1] S. Engblom. *Some thoughts on multiscale simulation of rigid fibers in Stokes flow.* Version 2, Uppsala University, 2011.

[2] A. Goude, S. Engblom. *Adaptive fast multipole methods on the GPU.* J. Supercomp., 63(3):897-918, 2013. doi:10.1007/s11227-012-0836-0

[3] Khronos Group. OpenCL. http://www.khronos.org/opencl/, May 2014.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, 5th Edition, 2011.

[5] Mathworks. MATLAB. http://www.mathworks.se/products/matlab/, May 2014.

[6] Mathworks. MEX. http://www.mathworks.se/help/matlab/matlab_external/introducing-mex-files.html, May 2014.

[7] Mathworks. Parallel Computing Toolbox. http://www.mathworks.se/products/parallel-computing/, May 2014.

[8] C. A. Navarro, N. Hitschfeld, L. Mateu. *A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures.* C. Comput. Phys., 15(2):285-329, 2014. doi:10.4208/cicp.110113.010813a.

[9] NVIDIA Corporation. CUDA. http://www.nvidia.com/object/cuda_home_new.html, May 2014.

[10] NVIDIA Corporation. GeForce GTX 260. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-260, May 2014.

[11] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide.* NVIDIA Corporation, Version 3.2, 2010.

[12] NVIDIA Corporation. *CUDA C Best Practices Guide.* NVIDIA Corporation, Version 3.2, 2010.

[13] The Open Group. Pthreads. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html, May 2014.

[14] OpenMP Architecture Review Board. OpenMP. http://www.openmp.org, May 2014

[15] A-K. Tornberg and K. Gustavsson. *A numerical method for simulations of rigid fiber suspensions.* J. Comput. Phys., 215(1):172-196, 2006. doi:10.1016/j.jcp.2005.10.028.

A System specifications

A.1 CPU and RAM

CPU:	Intel® Core™ i7-920 Processor, 2.66 GHz, 4 Cores
RAM:	12GB DDR3 1066MHz

Table A.1: CPU and RAM

A.2 GPU

Device name:	GeForce GTX 260
Global memory amount:	938803200 bytes (895 MiB)
Shared memory amount per block:	16384 bytes (16 KiB)
Registers per block:	16384
Warp size:	32 threads
Max threads per block:	512
Max size of each dimension of a block:	[512 512 64]
Max size of each dimension of a grid:	[65535 65535 1]
Constant memory amount:	65536 bytes (64 KiB)
Compute Capability:	1.3
Max 1D texture size:	8192 bytes (8 KiB)
Device Overlap:	Yes
Number of multiprocessors:	27
Concurrent kernels support:	No

Table A.2: GPU

A.3 Software

Operating system:	Scientific Linux release 6.3 (Carbon) 64-bit
MATLAB version:	R2012b
CUDA version:	4.2
Graphics driver:	NVIDIA driver 295.41

Table A.3: Software

B Measured times

Fibers	CPU-only	GPU-extension
100	00:01:13.837	00:00:34.416
200	00:04:52.105	00:02:32.538
300	00:11:19.697	00:06:09.682
400	00:21:07.461	00:11:50.685
500	00:35:15.053	00:20:47.070
600	00:51:52.628	00:32:16.822
700	01:14:07.393	00:48:22.786
800	01:42:33.829	01:08:19.533
900	02:17:58.762	01:34:38.071
1000	02:57:26.179	02:04:45.749

Table B.1: Total time for a whole simulation

Fibers	CPU-only	GPU-extension	GPU-extension (non-parallel)
100	00:00:38.667	00:00:02.301	00:00:02.296
200	00:02:17.683	00:00:08.686	00:00:08.760
300	00:05:13.586	00:00:16.523	00:00:19.005
400	00:09:19.273	00:00:22.908	00:00:33.823
500	00:14:32.880	00:00:29.911	00:00:52.449
600	00:20:44.343	00:00:39.372	00:01:15.883
700	00:27:43.019	00:00:49.844	00:01:43.256
800	00:35:59.191	00:01:02.347	00:02:14.364
900	00:45:20.314	00:01:16.812	00:02:50.708
1000	00:56:09.036	00:01:32.200	00:03:30.541

Table B.2: Total time for `greendotleg1` and `greenint`