Universiteit Leiden
Opleiding Informatica

GPU acceleration of arbitrary
precision N-body code

Name: Terry Zabel
Date: January 9, 2018
1st supervisor: Prof. Dr. Aske Plaat
2nd supervisor: Prof. Dr. Simon Portegies Zwart

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Graphics processing units (GPUs) have a wide variety of applications these days due to the speedup that can be achieved in multiple areas like multimedia, deep learning and in our case high performance computing. In this thesis we will describe and discuss the parallelization of arbitrary precision N-body code using a GPU. We focused on optimizing the sequential arbitrary precision N-body code *Brutus*[8] provided by Boekholt & Portegies Zwart. In the experiments we compared our parallel implementation to the sequential implementation *Brutus* by scaling the amount of bodies and the amount of bits arbitrary precision. In the results we show a speedup of up to 20 times faster with the lowest precision and scaling the amount of bodies. The results also show that the speedup becomes less when the amount of bits precision is scaled, but if the amount of bodies is high enough the highest precision of 512 bits we used, still shows speedup. In the end we also examine the influence of our implemented optimizations for the presented parallel version using a GPU. The optimization timings show that it is most important to make sure that memory accesses are coalesced.

Contents

1 Introduction **1**
1.1 Memory Usage . . . 1
1.1.1 Shared Memory . . . 1
1.1.2 Coalesced Memory . . . 2
1.2 CPU/GPU Overhead . . . 2
1.3 Generic Code . . . 2
1.4 Legitimization . . . 2
1.5 Overview . . . 3

2 Background Information **4**
2.1 N-body problem . . . 4
2.2 Arbitrary Precision . . . 4
2.3 Bulirsch-Stoer Algorithm . . . 4
2.3.1 Richardson extrapolation . . . 4
2.3.2 Rational function extrapolation . . . 5
2.3.3 Midpoint method . . . 5
2.4 Brutus . . . 6
2.5 GPU Architecture . . . 9
2.5.1 Single Instruction Multiple Thread strategy . . . 9
2.5.2 Streaming Multiprocessor . . . 10
2.5.3 Shared Memory . . . 10
2.5.4 Global Memory and Local Memory . . . 10

3 Related Work **13**
3.1 Memory Usage . . . 13
3.1.1 Shared Memory . . . 13
3.1.2 Coalesced Memory . . . 14
3.2 CPU/GPU Overhead . . . 14
3.3 Generic Code . . . 14
3.4 Arbitrary Precision . . . 15

4 Implementation **16**
4.1 Arbitrary precision . . . 16
4.2 Memory Usage . . . 16
4.3 CPU/GPU Overhead . . . 16
4.4 Generic Code . . . 17
4.5 Parallelization . . . 17
4.6 Optimization . . . 18
4.6.1 Optimized Extrapolation . . . 18
4.6.2 Minimizing Scratch Space . . . 18
4.6.3 Using Shared Memory . . . 18
4.6.4 Separated Meta-data . . . 19
4.6.5 Implement Stepsize to Operations . . . 19

5 Experimental setup **21**
5.1 Random generation of initial conditions . . . 21
5.2 Plummer generations of initial conditions . . . 21
5.3 Optimization timings . . . 22

6 Results **23**
6.1 Random generation of initial conditions . . . 23
6.2 Plummer generations of initial conditions . . . 27
6.3 Optimization Timings . . . 28

7 Conclusions **29**
7.1 Brutus on GPU . . . 29
7.2 Memory Usage . . . 29
7.3 CPU/GPU Overhead . . . 29
7.4 Generic code . . . 30
7.5 Summary . . . 30

8 Discussion and Future Work **31**
8.1 Discussion . . . 31
8.2 Future Work . . . 31

References **32**

1 Introduction

Recently interest in exploiting graphics cards for high performance computing has grown. The capabilities of graphics cards are increasing rapidly and therefore the application range is growing as well [12], [20]. In simulations of scientific phenomena, scientists require ever higher precision to approach a true solution, which is the solution calculated with infinite precision [8]. Implementing a parallel implementation takes time to divide the workload and combine the result. However, as the computational power and memory speed of parallel platforms grow, the time this overhead takes becomes less. In this thesis we will work on complex simulations with optimized GPU code and research the possibility of an efficient implementation of *Brutus*[8] on a GPU by parallelizing over the amount of bodies. Brutus uses arbitrary precision combined with the Bulirsch-Stoer algorithm to regulate how precise the calculated solution will be while making a trade-off against the time it takes to perform the calculations. When programming a GPU with performance as a priority, the CUDA programming guide addresses three basic strategies for overall performance optimizations in Chapter 5 [28]. From these guidelines we extract two points regarding the optimization of the application and one point regarding optimal settings for different compute capabilities. These three points are:

- Memory usage.
- CPU/GPU overhead.
- Generic code with respect to the GPU architecture.

Therefore our research questions are:

RQ1: *How to efficiently use GPU memory and deal with its limitations when using arbitrary precision on a GPU?*

RQ2: *How to minimize CPU/GPU overhead?*

RQ3: *How to make the code as generic as possible with respect to the GPU architecture?*

In the following sections we will elaborate on these research questions.

1.1 Memory Usage

In this section we will give an introduction on the two main techniques to overcome slow memory accesses using a GPU.

1.1.1 Shared Memory

First of all the amount of memory as well as the speed of the memory on graphics cards can be limiting, especially when working with arbitrary precision. In case of shared memory, which is a small amount of on-chip memory for fast accesses, that means that the maximum amount of shared memory per thread block on all compute capabilities is 48 KB which equals 1024 64-bit integers according to table 14 of the CUDA C Programming Guide [28]. When working with arbitrary precision the size of only one number can in our case already be up to 512 bits which makes it hard and possibly inefficient to use shared memory. Nevertheless we will try to use shared memory because the access time is low compared to global memory, which is explained in more detail in section 4.1.

1.1.2 Coalesced Memory

Besides using shared memory the performance of an application on GPU is also depending on coalescing memory accesses when accessing global memory on the GPU. This is a concept that is specific to GPU architectures where it matters a lot in which order memory accesses are done. We will describe this concept in more detail in section 3.1.2 using related work.

1.2 CPU/GPU Overhead

Working with a GPU does not mean only the GPU is used, it needs to be controlled by the CPU. This includes declaring memory in the GPU global memory beforehand, transferring initial values to the GPU, potentially declaring shared memory and getting results back from the GPU to be able to output them. In general being able to get speedup by using a GPU is dependent on this relatively slow but necessary communication. In our case however, there is not much output needed until the end of a run and it is possible to perform all the calculations using only the GPU as well. This means we can prevent having to transfer most of the data back and forth between the CPU and GPU. In general the performance gain from parallelizing the program should weigh up against the overhead it takes to accomplish the parallelization. Regarding that concept parallelizing using a GPU is not different from parallelizing using multiple CPUs.

1.3 Generic Code

Just as with CPUs the architectures of GPUs are different depending on the manufacturer and model. However with GPUs the programming language can also depend on the manufacturer. In our case we will focus on using the programming language CUDA[28] by NVIDIA and the different compute capabilities that come with this language. We made this choice because we have high-end NVIDIA devices available and CUDA can provide NVIDIA-specific features for increased performance where OpenCL[14] is not able to provide such a specific performance improvement. The compute capabilities of NVIDIA devices depend on the model and are related to different hardware specifications combined with different possibilities which can be found in Chapter G of the CUDA C Programming Guide [28]. In this project we will try to make the code as adaptable as possible to make it easier to optimize for different compute capabilities.

1.4 Legitimization

In the area of astronomy simulating N-body systems with complex behaviors require a high and known precision to approach a true result, which is the perfect solution with endless precision. Therefore N-body program Brutus [8] was made and used for generating predictions on movements in N-body systems. This program was made sequential for CPU and therefore not scalable for N-body systems with a large amount of bodies. It would take years to make precise predictions for a specified period of time when dealing with a large N-body problem [8]. In this work we strive to make the N-body program Brutus more scalable by using a GPU. We also make sure to provide the same possibilities the program Brutus provided for small N-body problems, by keeping the same high precision standard while making it feasible to work with large N-body problems.

1.5 Overview

In this thesis we will first give background information in Chapter 2, where we go into more detail about the concepts and methods we used. Second we will talk about related work in Chapter 3, where we showcase work of other people in the same research area. Third we will elaborate on our implementation in Chapter 4, where we explain with what idea we made our code and explain our optimizations. Fourth we have our experimental setup in Chapter 5, where we specify the hardware we used and go into more detail about the choices we made related to the experiments. Fifth we have the results of our experiments in Chapter 6, which showcase the outcomes of our experiments. Sixth we will go into conclusion in Chapter 7, where we will summarize and answer the research questions. Finally we will discuss our work and provide future work in Chapter 8, where we talk about the usefulness of our work and elaborate on possible further optimizations.

2 Background Information

In the introduction we mentioned arbitrary precision, n-body program *Brutus* and GPU architectures. In this chapter we will first provide background information for the n-body problem. Second we will give an introduction to arbitrary precision. Third we will elaborate in more detail about *Bulirsch-Stoer*[34], the base algorithm of Brutus. Fourth we will give an overview of Brutus itself. Fifth and last we will explain how a GPU works compared to a CPU and the general layout and memory model of a GPU.

2.1 N-body problem

The n-body problem is the problem of predicting the movements of a cluster of objects that are influencing one another by gravitational forces [24].

2.2 Arbitrary Precision

In this section we will give a short description of arbitrary precision and when it is used. Arbitrary precision arithmetic, also called multiple-precision arithmetic, is precision limited only by the amount of memory [21]. When using arbitrary precision one number is represented by multiple instances of a regular 8-64 bit data structure and typically stored in an array. In general arbitrary precision is used when the calculation time is not a limiting factor or when the precision of the calculated results is important.

2.3 Bulirsch-Stoer Algorithm

In this section we will go into detail about the Bulirsch-Stoer algorithm [34]. The Bulirsch-Stoer algorithm is an algorithm that uses three main concepts which we will explain in their own subsections below.

2.3.1 Richardson extrapolation

The idea of the Richardson extrapolation[10] is to calculate a chosen function for multiple different stepsizes and then extrapolate the results to an infinitely small stepsize and thus and infinite amount of steps. A visualization of this Richardson extrapolation is shown in Figure 1.

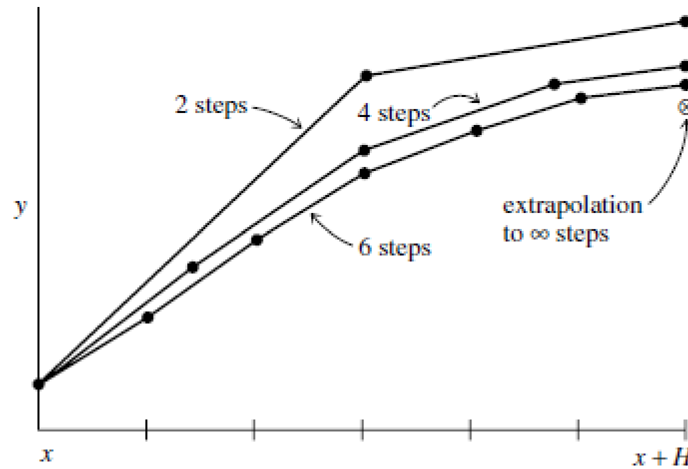


Figure 1: Visualization of the Richardson extrapolation [3].

2.3.2 Rational function extrapolation

Instead of doing a Richardson extrapolation with a regular polynomial function, Bulirsch and Stoer discovered that using a rational function as a fitting function is superior. A polynomial extrapolation generally only works well when the distance between the known data and the nearest pole is rather large. Rational extrapolations however can also be accurate for nearby poles.

2.3.3 Midpoint method

The midpoint method[9] is a second-order method to solve differential equations based on the first-order Euler method. We will not go into detail about this method, but we will note the fact that the accuracy of the solution increases two orders whenever the result of a previously calculated attempt is used. This is useful in the Bulirsch-Stoer algorithm because the Richardson extrapolation is done for different stepsizes and therefore has multiple attempts already.

2.4 Brutus

Brutus is an N-body [25] program using arbitrary precision combined with the Bulirsch-Stoer algorithm described above [8]. By setting the tolerance in the Bulirsch-Stoer algorithm we can specify how precise our outcome will be with a 100% certainty. We started by making a workflow overview and a UML diagram of Brutus to get a basic idea of how the program is build for CPU. The workflow overview is shown in Figure 2 and the UML diagram is shown in Figure 3. These figures show the four classes of Brutus, which we discuss in more detail below:

- Brutus: The base class of the program, instantiated in the main function and responsible for the outer loop that executes timesteps until the given end time for that cycle is reached or conversion is not possible within the given limits.
- Cluster: The class holding one or more stars and performing calculations using those stars.
- Bulirsch-Stoer: The class performing the Bulirsch-Stoer algorithm on a given Cluster using a given timestep.
- Star: The class holding the parameters of the bodies, instantiated and inherited by the Cluster.

NB: the class Brutus in this section should not be confused with the program Brutus it is part of.

Figure 2 shows the basic workflow of Brutus. After initialization of the program, the main loop is started which runs until the given end time is reached or until an early exit occurred. Within the main loop an instance of the class Brutus evolves with a given timestep. This evolve will keep going until the larger timestep given by the main loop is reached. Within the evolve loop the class *Bulirsch-Stoer* integrates over the timestep given by the evolve using the function *step* of the class *Bulirsch-Stoer*. This will continue until the result of the extrapolation is accepted or the maximum amount of given tries is reached while dividing the timestep by 2 every try. Within the function *step* of the class *Bulirsch-Stoer* an instance of *Cluster* will be created. With an increasing amount of clusters and steps every try, the function *step* of the class *Cluster* will be executed, which does the N-body calculation using arbitrary precision. Then the extrapolation will be done over the generated clusters and the error difference between the clusters generated with different size and amount of timesteps will be calculated. This will continue until the calculated error is smaller than the given tolerance and therefore accepted or the set amount of maximum tries is reached.
When the maximum amount of given tries for the outer loop is reached without an acceptable result, it will cause an early exit of the program. If the result is accepted it will be propagated back to the main loop and continue with the next timestep until the program reaches the given end time. The result will be printed to a file every timestep and after the main loop finishes when the end time is reached the final result will be printed to the same file as well.

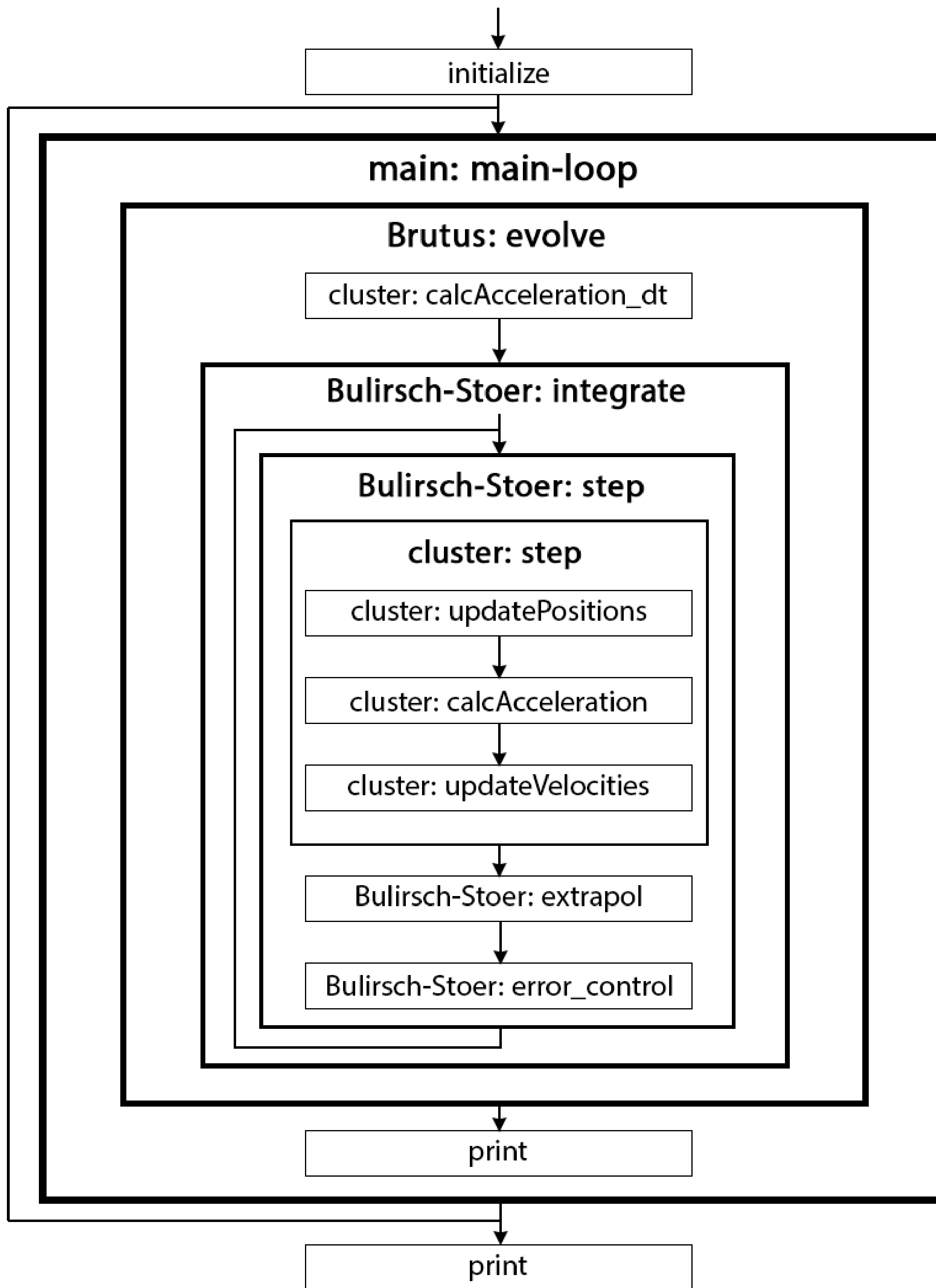


Figure 2: Basic workflow of Brutus.

Figure 3: UML diagram of Brutus.

2.5 GPU Architecture

The architecture of a GPU is different from the architecture of a CPU. Figure 4 shows the most important difference. A CPU consists of a few high performance cores able to perform sequential calculations very fast. On the contrary a GPU at this time can have up to 4000 lower performance cores grouped together in a *warp* with one control unit per warp. This makes a GPU not only excellent for performing fine-grain parallelization for example in vector/matrix calculations in computer graphics, but also for other problems that have a lot of concurrent calculations. Besides the obvious difference in amount of cores there is also a difference in execution strategy and memory hierarchy. In the following part we will describe the single instruction multiple thread strategy (SIMT) that is used, show the general layout of a GPU and elaborate on the memory options that are available when using GPUs.

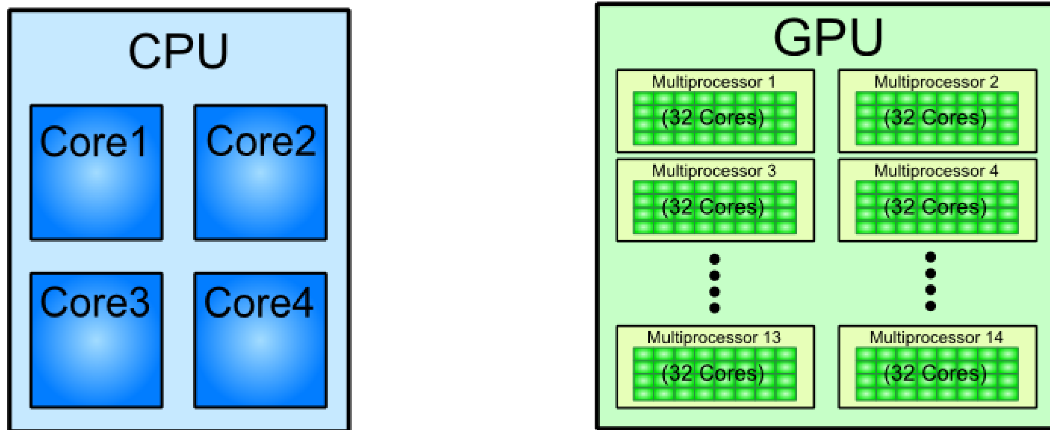


Figure 4: CPU vs GPU Architecture [1].

2.5.1 Single Instruction Multiple Thread strategy

The single instruction multiple thread strategy (SIMT) is a method for executing threads. Just as parallelizing with a CPU, parallelizing with a GPU works with threads that can be declared by the programmer. However, because there are so many cores on a GPU, for efficiency not every core can have its own control unit. Therefore 32 threads are grouped together in a warp and mapped on 32 cores. Every warp has its own streaming processor (SP) which controls the execution of the instructions. Using SIMT means that all of the 32 threads in the same warp execute the same instruction at the same time. A disadvantage of this is that if threads in the same warp take different paths in a branch, one thread will be waiting for the other thread to execute their part of the branch and then the other way around. Therefore programming branches on a GPU must be done carefully.

2.5.2 Streaming Multiprocessor

Figure 5 shows four warps in the layout of a streaming multiprocessor (SM) of the Maxwell architecture. Although the exact implementation can differ depending on the specific architecture that is used, the layout of a SM will be the same. When using NVIDIA GPUs every architecture comes with its own compute capability from which all the properties can be found in Table 13 of the CUDA Programming Guide [28]. This is the point where the software implementation starts to diverge from the hardware implementation. The programmer has to group threads into blocks where the hardware schedules warps onto SMs. So for example the programmer invokes a kernel with 10 blocks each consisting of 50 threads. For every block 32 threads will be assigned in a warp and then the other 18 threads will be assigned to another a warp. In this case it means that for every block there will be a warp using only 18 out of the possible 32 threads. Since all architectures and thus compute capability versions of CUDA have 32 concurrent cores per warp, it is efficient to have the amount of threads per block in multiples of 32.

2.5.3 Shared Memory

As shown in Figure 5 every streaming multiprocessor has its own shared memory. From the CPU programming point of view shared memory on a GPU can be seen as an explicitly managed cache. If parts of global memory are going to be used multiple times it is efficient to load them into shared memory first. If it is possible all necessary data should be copied in shared memory at once, but if the necessary data does not fit in the shared memory it is necessary to load parts of the needed data in the shared memory at a time. This has to be done manually by writing an own algorithm depending on the application and is only efficient if the part that is loaded into the shared memory is used multiple times before the next part is loaded into the shared memory. Unlike global memory and local memory, shared memory of a GPU is on-chip, which makes it a lot faster than the other implementations and highly preferred to use when performance is a priority.

2.5.4 Global Memory and Local Memory

Besides the on-chip shared memory we already mentioned global and local memory. Figure 6 gives an overview of the memory hierarchy in NVIDIA GPUs. The *per-thread private memory* that is shown in the overview is another name for the local memory. What is not obvious from the figure is that as mentioned before the shared memory is on-chip and the local and global memory are not. Because local and global memory are not on-chip accesses to these memories are expensive and should be optimized and minimized. The best way to optimize is by making sure memory accesses are coalesced. This can be done in three different ways; access patterns, using data types that have the right size and alignment and data padding. Access patterns are dependent on the architecture and thus the compute capability that is used and are described in more detail in the CUDA Programming Guide [28]. Data types with the right size and alignment are data types consisting of 1, 2, 4, 8 or 16 bytes and their address as a multiple of that size, also called *naturally aligned*. Data padding can be useful when the size of an array is not a multiple of the warp size. Memory accesses will be much more efficient if the array size is rounded up to the closest multiple of the warp size and to accomplish that the data has to be padded.

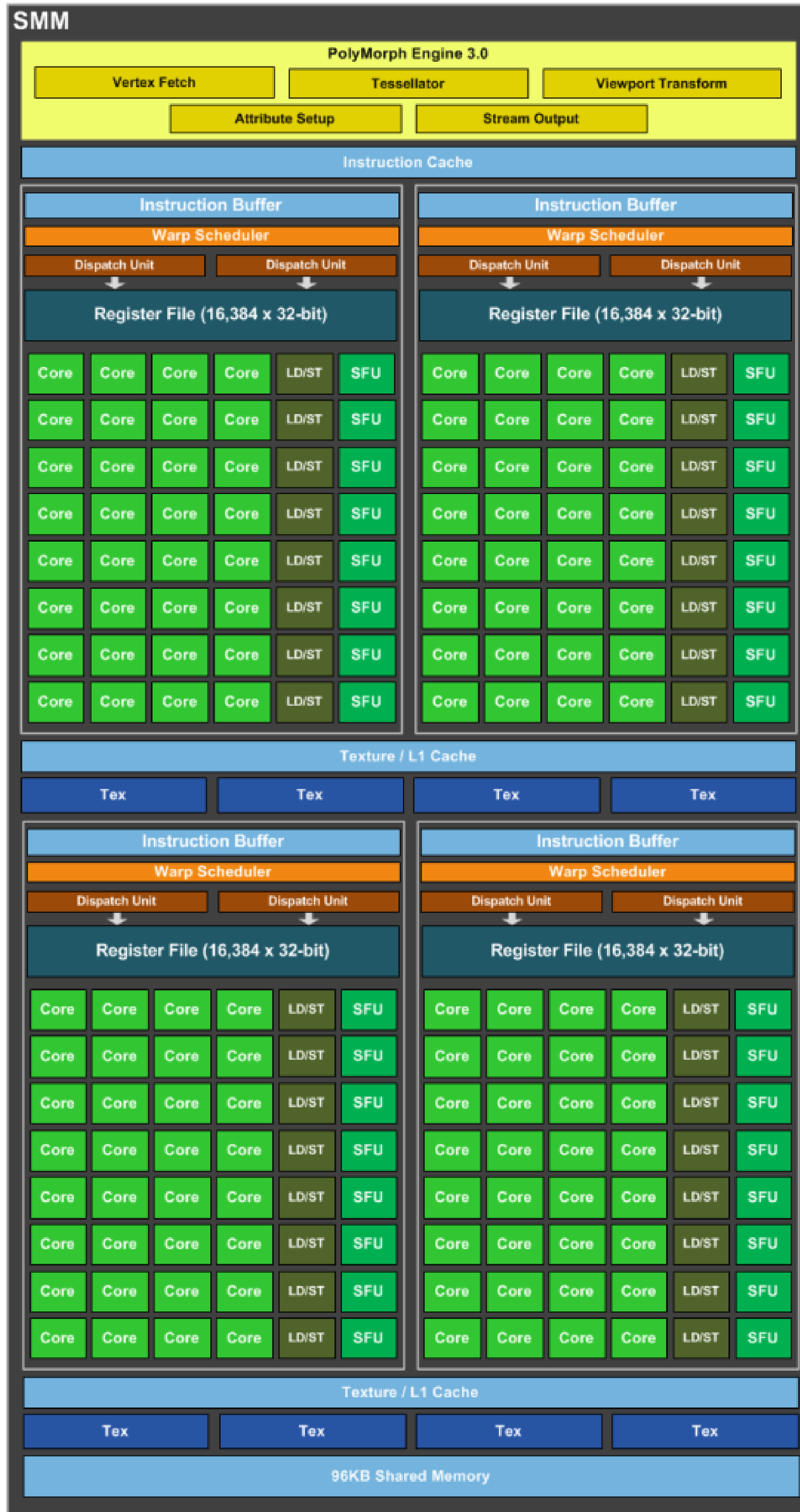


Figure 5: Streaming Multiprocessor [29].

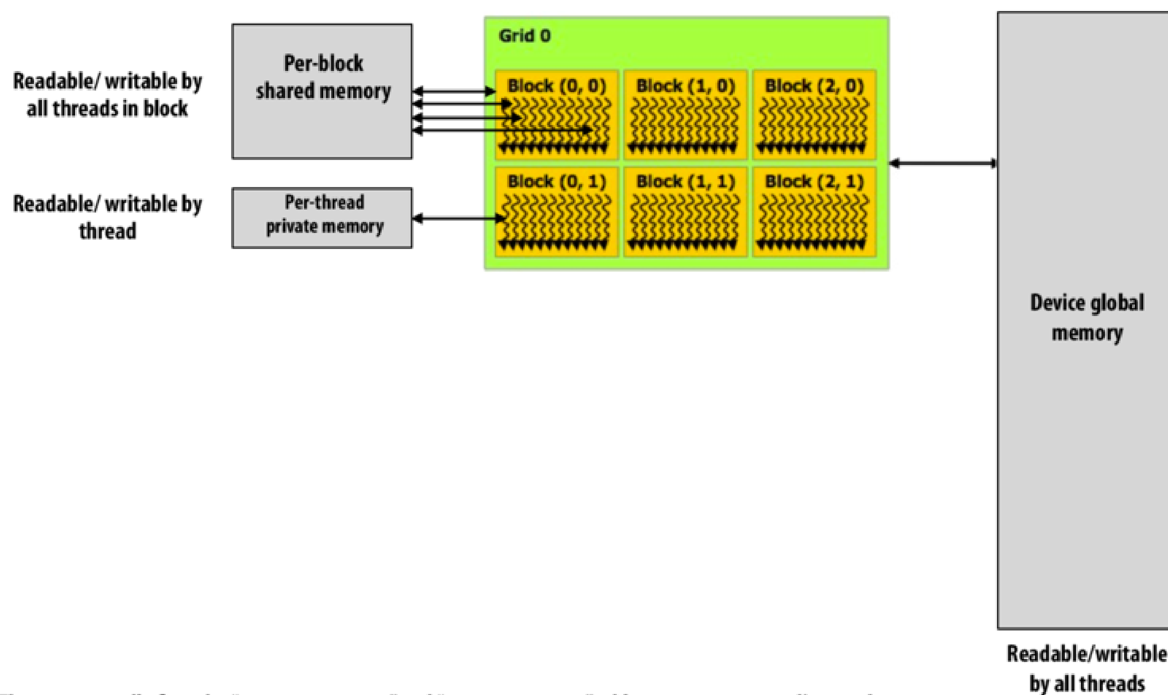


Figure 6: Streaming Multiprocessor [33].

3 Related Work

We will use the same structure in this section to elaborate on related work that is done on GPU memory usage, CPU/GPU overhead and making generic code with respect to the GPU architecture. Besides we will also mention work related to arbitrary precision. When programming a GPU it is important to optimize the code to be able to get as much performance gain as possible. In 2008 S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk and W. Hwu have made an overview of optimization principles that can be used to accomplish this task [32]. Specifically they mentioned that key to performance when using a GPU is using massive multithreading to utilize all the available cores and hide global memory latency when using shared memory is not possible. However there can also be more application specific issues that need to be optimized or need to be taken care of.

3.1 Memory Usage

An analytical model has been made that considers memory parallelism on top of the standard parallelism [16]. This shows how important the way memory is accessed is for GPUs. This is further specified into using shared memory and making sure global memory accesses are coalesced.

3.1.1 Shared Memory

On-chip shared memory provides low latency, high-bandwidth access to data shared by cooperating threads in the same CUDA thread block. Fast shared memory significantly boosts the performance of many applications having predictable regular addressing patterns, while reducing DRAM memory traffic [27]. To be able to use GPU shared memory when the used data is larger than the maximum shared memory size an algorithm has to be made to load parts of the global memory in the faster shared memory. However, this is only useful when the loaded memory is used more than once, since an initial load from the global memory is always necessary. An equivalent problem occurs when a GPU is used for doing calculations on large graphs, elaborated on by Pawan Harish and P. J. Narayanan [15]. In this case, the used graph does not only not fit in the shared memory, but also does not fit in the global memory. Although global memory sizes of GPUs have greatly increased over the passed years, the shared memory sizes are still the same. This means that the problem of limited global memory described by Harish [15] has become less of a problem, but using shared memory efficiently is still a major issue while handling large amounts of data on a GPU. The origin of this problem lies in the physical architecture of a GPU, which is described in more detail in section 4.1.

3.1.2 Coalesced Memory

As mentioned before shared memory is accessed with full control by the programmer, which means that the programmer has direct influence on the alignment of the memory accesses by deciding the execution order. Making sure global memory accesses are coalesced is a much more complicated issue, because the memory has to be structured without having direct control over the order of the memory accesses. According to the CUDA Programming Guide [28] influence on performance of non-coalesced memory accesses instead of coalesced memory accesses depends on the compute capability that is used, but is always important to maximize memory throughput. This also means that depending on the compute capability the optimal memory storage patterns are different. Fortunately NVIDIA has detailed explanations on when and how coalesced memory accesses can be used for the different compute capabilities, which can be found again in their comprehensive CUDA Programming Guide in section G [28]. To emphasize the importance of avoiding non-coalesced memory accesses a fundamental study to the problem has been done and a feasible way to minimize non-coalesced memory accesses for a variety of irregular applications has been provided [36].

3.2 CPU/GPU Overhead

Communication overhead is just as important when parallelizing using a GPU as when parallelizing using CPUs. To not only minimize, but also simplify this task, an automatic CPU-GPU communication manager and communication optimizer has been made in 2011[18]. However, the results in this article show that in the cases where a manual implementation was available this implementation is in all cases faster than the automated and optimized implementations. A year later in 2012 they present the first automatic system to manage complex and recursive data-structures without static analyses [17]. By replacing static analyses with a dynamic run-time system, their presented system *DyManD* overcomes the performance limitations of alias analysis and enables management for complex and recursive data-structures. They magnify the problem that CPU/GPU overhead can be tedious and error-prone. Therefore their presented system can be much better than a manual implementation, but again their results show that it is not always better than the manual implementation.

3.3 Generic Code

Since there are different GPU hardware platforms which support specific languages there has been a challenge making one language that supports all platforms. OpenCL is trying to accomplish this task, but focuses on programming portability rather than performance portability [11]. This logically leads to a comparison between the platform specific programming tools like CUDA and the portable programming tools like OpenCL. For example K. Karimi, N. Dickson, F. Hamze have compared CUDA and OpenCL using near-identical kernels from a Quantum Monte Carlo application and show that OpenCL indeed has less performance than CUDA [19]. A similar result is shown in the work of G. Peterson, A. Gothandaraman, R. Weber and R. Hinde [30]. They not only compared CUDA vs OpenCL, but also added C++ using multicore processors and a VHDL implementation running on a Xilinx FPGA. They also illustrate that graphics accelerators can make simulations involving large numbers of particles feasible.

3.4 Arbitrary Precision

Multiple arbitrary precision libraries are already existent for CPU [21]. However, for GPU the focus has been on speeding up arbitrary precision to high amounts of bits by using the GPU [22],[23]. The disadvantage of this is that the GPU is already used by the arbitrary precision library, where we want to use the GPU for a higher level parallelization which uses arbitrary precision operations on a lower level. The only library we could find that provides the mentioned requirements is CUMP [26]. The problem with CUMP is that it is based on CPU arbitrary precision library GMP [2] rather than the arbitrary precision library MPFR [13] which is used by Brutus. GMP and MPFR use different rounding methods and therefore we cannot use CUMP if we want to be similar to Brutus.

4 Implementation

In this chapter we will describe and explain the way we implemented Brutus [8] on GPU and elaborate on the optimizations we did.

4.1 Arbitrary precision

As we mentioned already, the only arbitrary precision library that existed for GPU was CUMP [26], which was published in 2011. CUMP is based on arbitrary precision library GMP for CPU [2]. Brutus uses MPFR [13] as arbitrary precision library for CPU, which uses different rounding methods than GMP. Because of the age of CUMP and the different rounding methods, we decided to port MPFR to GPU by rewriting the operations we need into plain C++ functions. The majority of the rewriting was done by S. Sultan [1]. The advantage of this is that we have the ability to adapt and optimize these operations not only for using a GPU, but also for our specific application.

4.2 Memory Usage

Since global memory of state-of-the-art GPUs is large enough to not be limiting for us, we mostly have to worry about how efficient we use the global memory rather than how much global memory we use in total. Using only global memory results in relatively slow memory accesses compared to using shared memory, but is not always possible. In section 5.6.3 we will explain why in our case we are bound to using only global memory. At first we implemented the same memory structure as we would on CPU, where all values are in a logical order with the same size and easy to access. However, this is far from optimal when using a GPU and parallel memory accesses will be non-coalesced. In section 5.6.2, 5.6.4 and 5.6.5 we will elaborate which changes we made to optimize our basic memory layout for GPU and show how much performance was gained which each optimization.

4.3 CPU/GPU Overhead

To minimize memory overhead between the CPU and the GPU we load the initial cluster on GPU once and declare other necessary memory only on the GPU. By keeping all memory on the GPU the only memory transfers that need to happen are loading of the initial cluster to the GPU and transferring the resulting cluster back to the CPU. Keeping all memory on the GPU means we have to do all functions that operate on the data on the GPU. The only possible problem with this approach would be that not all functions we use are as well parallelizable as the calculation of the accelerations. However, this is not a problem for us, because calculating the accelerations and then reducing them to one value per body is taking 99% of the time of our program according to measurements with the NVIDIA profiler [7]. The performance gain for the other functions if we would run them on the CPU is in the first place percentage-wise not interesting. In the second place the performance gained by doing these functions using the CPU would most likely be less than the performance lost by the data transfers that would be necessary from and to the GPU to be able to run these functions using the CPU.

[1] shabaz.sultan@gmail.com

4.4 Generic Code

As we already mentioned in section 1.3, we have focused on implementing in CUDA, because we have high-end NVIDIA devices available and CUDA has more possibilities than OpenCL when looking for performance in the case of NVIDIA cards. Our current implementation is compatible with CUDA compute capability 2.X and higher, since we are using new and delete for our own data structure in the form of a C++ class and declaring global memory in device code, which require both at least CUDA compute capability 2.X [28]. We have optimized our code for compute capability 6.0 since we have the opportunity to use one of the NVIDIA Tesla P100 cards of little green machine II [4]. Unfortunately we did not have the time to implement and optimize support for all compute capabilities, but we have set up all related values as class or global variables.

4.5 Parallelization

For the calculation of the accelerations we parallelized over the number of bodies. This means we have a separate thread for every combination of two bodies and calculate the accelerations for every axis between them. We store all those accelerations and reduce them to one acceleration value per axis per body using a tree, which results in $N * 3$ arbitrary precise values to store the accelerations.
When updating the velocities and positions we parallelize over the amount of bodies and the three axes. This means we have a separate thread for every axis for every body.
For the extrapolation we parallelize over the amount of bodies and the six variables of the bodies that get updated. These six variables are the x-, y- and z-position and the x-, y- and z- acceleration.
The compute capability we have is 6.0 for a Tesla P100. For this compute capability the maximum amount of resident blocks per multiprocessor is 32, the maximum amount of resident warps per multiprocessor is 64 and the maximum amount of resident threads per multiprocessor is 2048. This means that to reach the maximum amount of resident threads per multiprocessor at least $2048/32 = 64$ threads per block are necessary. For this reason the performance will not improve when the amount of threads per block increases. However, if the amount of threads per block increases it decreases the total amount of blocks which means it reduces the amount of SMs it can use at the same time, since all threads of one block have to run on the same SM. In our case it means that if the amount of bodies decreases, the amount of total blocks decreases and therefore the total occupancy of the GPU will decrease if the amount of blocks gets less than the amount of SMs. In conclusion this means that we want to have as much blocks as possible, but with full occupancy for every SM, which is 64 blocks per thread in our case.

4.6 Optimization

To speed up the initial working version of Brutus on GPU we tried the following optimizations:

- Rewriting the extrapolation, so it uses previous calculations instead of recalculating everything.
- Minimizing amount of scratch necessary for the operations.
- Using shared memory for the operations.
- Separating meta-data from the array of numbers.
- Implementing a stepsize for all operations.

An overview of the effects of these optimizations on the calculation times for different amounts of bodies will be given and elaborated on in the results below.

4.6.1 Optimized Extrapolation

The extrapolation used in the CPU version described in background information has been optimized by previously mentioned S. Sultan, who did the majority of the work of rewriting MPFR to plain C++. This optimization consists of storing previous results and only a single loop is necessary to do the extrapolation compared to the original, which uses a double loop to do the extrapolation. This optimization reduces the complexity of the extrapolation from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. Although previous extrapolations are being stored, the amount of memory used does not increase because first the previously calculated clusters were stored instead of the previously calculated extrapolations. In our experiments we will compare our GPU version to the optimized CPU version.

4.6.2 Minimizing Scratch Space

To minimize the scratch space that is necessary to do the calculations with arbitrary precision, we will not add the scratch space to every number in the cluster. Instead we use temporary variables with scratch space to do the calculations and write the answer back to the right place afterwards. Besides we make sure that the memory accesses to these temporary variables are coalesced, which we describe in the last two optimizations.

4.6.3 Using Shared Memory

For doing the calculations we tried using shared memory for faster memory accesses. The problem is that we are using arbitrary precision and need about eight times the size of the number as scratch space for calculating the inverse square root. Therefore the amount of available shared memory per streaming multiprocessor (SM) is not high enough to make using shared memory efficient. Because every block would need the full shared memory of a SM to only run a few threads the occupancy of the GPU is too low to be compensated by the faster memory accesses. So in the end it is faster in our case to use global memory and make sure the accesses to that global memory are as efficient as possible by coalescing as much accesses as possible.

4.6.4 Separated Meta-data

The meta-data we use for our operations are the sign, exponent and zero flag. The zero flag is a separate indicator which defines the arbitrary precision number to be zero if set to true. We store those values in one unsigned long to save memory. When we access the meta-data of multiple numbers in parallel we want to make sure those memory accesses are coalesced, which means we want them next to each other in the memory. The only way to accomplish this is to separate the meta-data from the array of limbs that represent the arbitrary precision number. This means we have to provide the memory address of the meta-data and the array of limbs to every operation instead of one memory address to the entire arbitrary precision number. Figure 7 illustrates this separation.

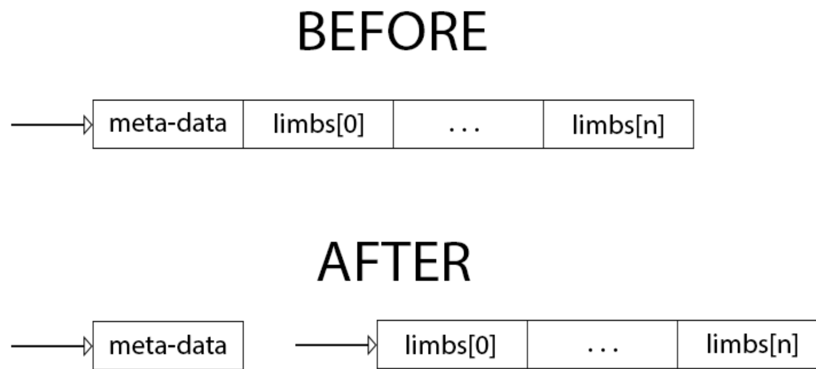
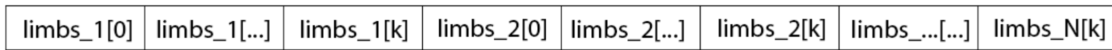


Figure 7: Illustration of the separation of the meta-data from the array of limbs.

4.6.5 Implement Stepsize to Operations

For the same reason as we separated the meta-data, we want to be able to manage the memory for the separate limbs. We make the accesses of the separate limbs more efficient by making sure the limbs that are accessed in parallel are next to each other in the memory. To accomplish this we have implemented a stepsize for every operation. Using this stepsize we are able to put all limbs with the same index of a group of parallel used arbitrary precision numbers next to each other. Figure 8 shows the implementation of the memory for a group of N arbitrary precision numbers. The operations are accessing the right limbs by using a stepsize of N when iterating over the indices of the limbs: $limbs[i * stepsize]$ instead of $limbs[i]$. This works well when the operations use the same size for all the limbs during the calculations. However, at some point the division operation converts the 64-bit memory address to a 32-bit one, which means we get the memory layout shown in Figure 9. To access the correct 32-bit part of the limb we will use $limbs[(i/2) * stepsize + i\%2]$ instead of $limbs[i]$. Note that the stepsize in this calculation is doubled since every 64-bit number converses to two 32-bit numbers.

BEFORE



AFTER

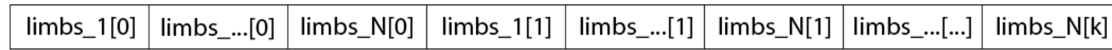


Figure 8: Memory layout of a group of N arbitrary precision numbers with k limbs per number.

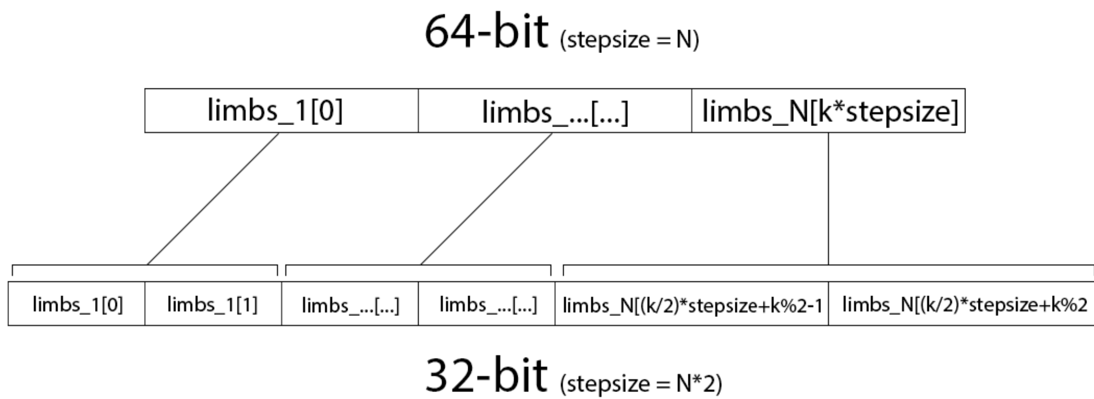


Figure 9: Conversion of a 64-bit array of limbs to a 32-bit array of limbs.

5 Experimental setup

In this chapter we will elaborate on the hardware and configurations we used for our experiments. As we already briefly mentioned, we were allowed access to Little Green Machine II (LGM2) [4]. This machine consists of an IBM Power 8[35] processor and sixteen NVIDIA Tesla P100 GPUs. We used this machine to run our optimized CPU version using the IBM Power 8 processor as well as to run our optimized GPU version using the IBM Power 8 processor in combination with one of the NVIDIA Tesla P100 GPUs. We have tested for two different types of initial conditions which we will explain in more detail.

5.1 Random generation of initial conditions

In this section we will go into more detail about the setup for the timings with complete random initial conditions. In this setup the initial conditions have been generated by generating random values between 1-1000 for every variable[2]. Then we scaled for either the amount of bodies or the amount of bits precision. For the amount of bodies we scaled from 10 up to 1000. For the amount of bits precision we scaled from 64 to 512 with steps of 64 bits, because as we mentioned before every limb consists of 64 bits. For example this means with a precision of 65 bits we will use two limbs and therefore 128 bits of memory for every number, which is the same amount of memory as when using a precision of 128 bits. Besides the amount of bodies and the precision we also set the start time, end time, stepsize and tolerance. We used a time frame of 0 to 5 with steps of 1 and an infinite Bulirsch-Stoer tolerance. This way we are sure that we will do exactly 5 cycles of calculations for both the GPU and the CPU, so we have a fair comparison for the timing. We time the wall clock time just before evolving and after the evolution of the cluster, so we only time the actual calculations. Using this setup we will be able to calculate the speedup of purely the calculations comparing the optimized CPU version to the optimized GPU version. On the other hand this does not provide a realistic example, so we will run with another setup which we will explain in the next section.

5.2 Plummer generations of initial conditions

In this section we will elaborate on the more realistic setup where we use random initial conditions generated according to the Plummer model[31] with an equal mass for every body in the sphere. To make it more realistic, but still feasible we set a tolerance of $1e{-2}$ and adjusted the time frame for the different amounts of bodies. Again we divided the time frame over 5 steps, but this time the result will have to meet the set tolerance before going to the next step. This results in the following three smaller setups:

- 100 bodies with a time frame from 0 to 1 and a stepsize of 0.2.
- 300 bodies with a time frame from 0 to 0.5 and a stepsize of 0.06.
- 1000 bodies with a time frame from 0 to 0.1 and a stepsize of 0.02.

All three of the above smaller setups use 10 different random initial conditions that are generated according to the Plummer model [31] that we already mentioned above.

[2]Mass, X-position, Y-position, Z-position, X-velocity, Y-velocity and Z-velocity.

5.3 Optimization timings

In this section we will elaborate on the timings we did for the optimizations we described in section 4.6. For every optimization we will do a timing for a low amount of bodies and bits precision and a timing for a high amount of bodies and high amount of bits precision. We will time just one timestep for an infinite Bulirsch-Stoer tolerance, to make it feasible to run the least optimized version. We make a difference between a high amount of bodies and a high amount of bits precision, because different optimizations affect different aspects of the program. Just as with the timings described in section 5.1 we will use completely random initial conditions by generating random values between 1-1000 for every variable[3] Also we again use an infinite Bulirsch-Stoer tolerance to make sure the different optimizations will do exactly the same amount of work.

[3]Mass, X-position, Y-position, Z-position, X-velocity, Y-velocity and Z-velocity.

6 Results

In this chapter we will show and elaborate on the experiments that we described in chapter 5. These include:

- An experiment with completely random initial conditions using an infinite Bulirsch-Stoer tolerance.
- An experiment with random initial conditions according to the Plummer model[31] and a tolerance of 1×10^{-2}.
- An experiment that shows the performance gained by the optimizations we explained in section 4.6. This experiment uses the same configuration as the first one with completely random initial conditions and an infinite Bulirsch-Stoer tolerance.

6.1 Random generation of initial conditions

In this section we will elaborate on the results of the experiment described in section 5.1. First we have compared the wall clock time of the optimized CPU version to the optimized GPU version for five timesteps and an infinite Bulirsch-Stoer tolerance with an arbitrary precision of 64 bits. Figure 10 shows that our optimized GPU version already is faster for 50 bodies. Figure 10 also shows that our optimized GPU version keeps getting faster relative to the CPU until about 300 bodies where it is about 20 times faster than the CPU. At that point the GPU is at its full capacity and starts to scale in the same logarithmic way as the CPU. This means that for 1000 bodies the GPU still is about 20 times faster for this configuration. This is a result of having close to 4000 GPU cores instead of one CPU core, where the single CPU core has four times the clock speed of the GPU core. Theoretically this could result in a speedup of almost a factor 1000. Due to memory overhead, lower memory speed of the GPU and not accomplishing an occupancy of the GPU of 100% this results in only 20 times speedup. It was not possible to get 100% occupancy because of the amount of registers used by an arbitrary precision calculation combined with the limited amount of registers per streaming multiprocessor (SM). Therefore the number of parallel warps per SM could not be maximized.
Second we have taken the amount of bodies where our optimized GPU version gets an order of magnitude faster, which is around 100 bodies. We also examine the first optimal point for the GPU, which is around 300 bodies and the largest amount of bodies input we have, which is 1000 bodies. For these interesting amounts of bodies we scale the amount of bits precision. Figure 11, 12 and 13 show that in all cases our optimized GPU version is affected more by scaling the amount of bits arbitrary precision than the optimized CPU version. The higher efficiency of the CPU when handling a higher amount of memory accesses likely causes this difference.
For 100 bodies Figure 11 shows that for the highest measured bits of arbitrary precision our optimized GPU version is already starting to be slower than the optimized CPU version. However, Figure 12 and 13 show that the influence of scaling the amount of bits arbitrary precision does not scale linearly with the amount of bodies. This means that the higher the amount of bodies are used, the less influential the costs scaling the amount of bits arbitrary precision is relative to the total time. In short scaling the amount of bits arbitrary precision has a higher influence on the performance of our optimized GPU

version compared to the optimized CPU version. However, this influence becomes less important when the amount of bodies becomes higher.
Besides this comparison of GPU and CPU we also made a comparison between using the full GPU and using one GPU core. We enrolled the parallelism in for-loops while using one block and one thread. After that we ran the program for 300 bodies, an arbitrary precision of 64 bits, an infinite Bulirsch-Stoer tolerance and only one timestep. This took 1936212 milliseconds compared to the 515 milliseconds it took for the full GPU. The difference is almost a factor 3800, while using 3840 cores. Although this looks like a near perfect parallelization, the speedup is not only accomplished by the parallelization over the 3840 cores, but also by latency hiding. Latency hiding means that while threads are taking time to access the memory, other threads are using the cores to do their calculations. Therefore the time that one core with one thread normally has to wait for a memory access before doing the next calculation, is now used by another thread while the first thread is accessing the memory.
In conclusion these results show a speedup of up to 20 times for the lowest precision of 64 bits. This speedup becomes less when scaling the amount of bits arbitrary precision. For 100 bodies the GPU implementation becomes slower than the CPU implementation when using the highest amount of bits arbitrary precision. For 300 bodies the GPU implementation is still faster than the CPU implementation, but only 1.4 times for the highest amount of bits arbitrary precision. Finally for 1000 bodies the speedup of the parallel version is still 5 times when using the highest amount of bits arbitrary precision.

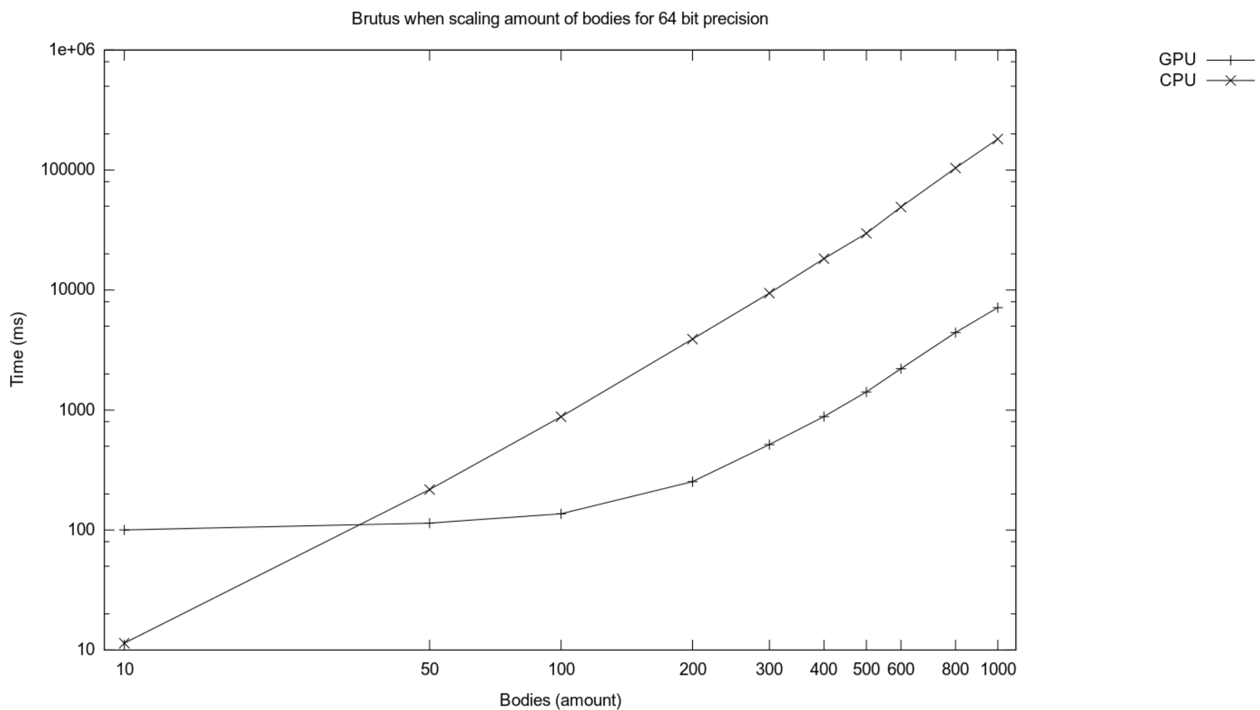


Figure 10: Calculation time of 5 timesteps with infinite Bulirsch-Stoer tolerance for different implementations of *Brutus* when scaling the amount of bodies with 64 bits precision, averaged over 10 runs.

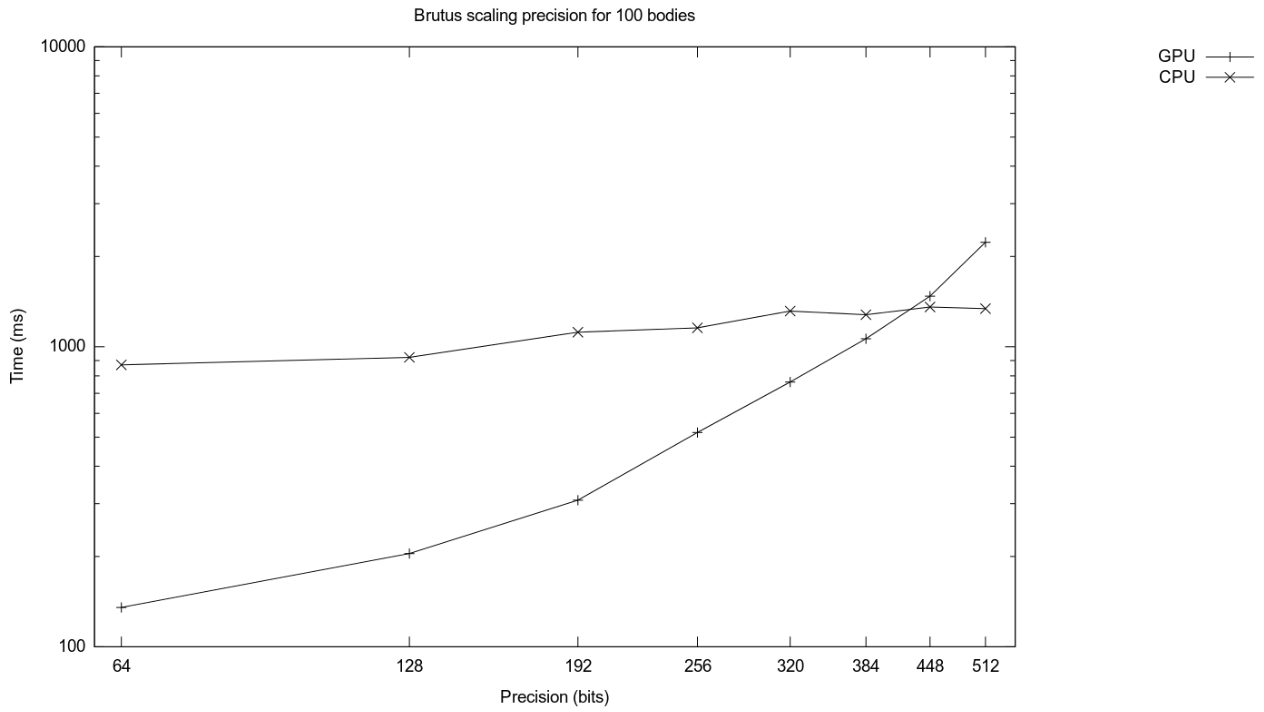


Figure 11: Calculation time of 5 timesteps with infinite Bulirsch-Stoer tolerance for different implementations of *Brutus* when scaling the precision for 100 bodies, averaged over 10 runs.

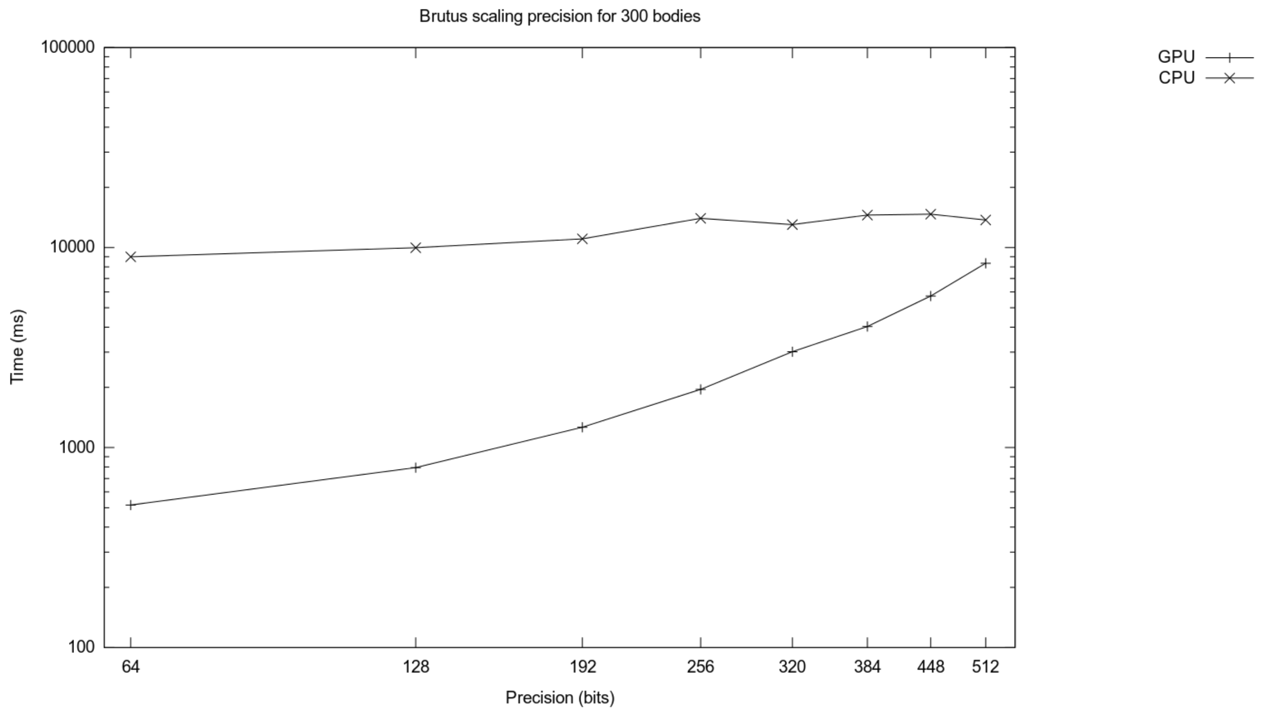


Figure 12: Calculation time of 5 timesteps with infinite Bulirsch-Stoer tolerance for different implementations of *Brutus* when scaling the precision for 300 bodies, averaged over 10 runs.

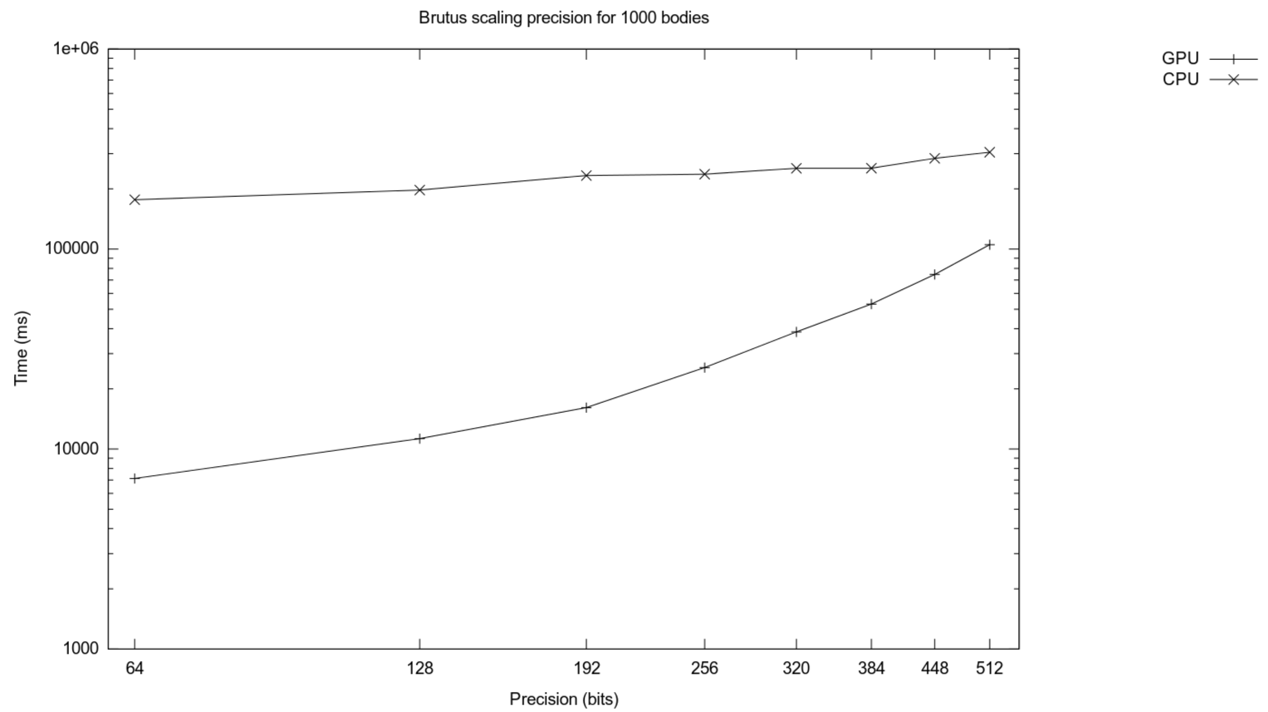


Figure 13: Calculation time of 5 timesteps with infinite Bulirsch-Stoer tolerance for different implementations of *Brutus* when scaling the precision for 1000 bodies, averaged over 10 runs.

6.2 Plummer generations of initial conditions

In this section we will elaborate on the results of the experiment described in section 5.2. We have made 10 different initial conditions again for 100, 300 and 1000 bodies. As mentioned before these initial conditions are generated according to the Plummer model[31]. For all these initial conditions the calculations of five timesteps of different size depending on the amount of bodies have been done. Further details are explained in section 5.2. Table 1 shows the overview of the time in milliseconds it took to do the calculations for every separate initial condition and an average of the initial conditions with the same amount of bodies. The timings in this table show that the differences in time between the optimized CPU version and optimized GPU version are different from the exact measurements we found in section 5.1. From this we can conclude that to actually calculate on the bodies with a tolerance there is not a general optimal set of input parameters that works for every initial conditions of the same size.
For example calculating five timesteps with 64 bits of arbitrary precision might be slower than calculating the same five timesteps with 128 arbitrary precision. This is the case, because the 64 bits version might have to make the timestep a lot smaller than the 128 bits version and therefore has to do a lot more cycles in the Bulirsch-Stoer algorithm to accomplish the required tolerance.
Also starting with a smaller timestep might influence the time it takes for the calculations to finish the same time frame although it takes more steps in total. For example, if the timestep is changed from 1 to 0.5 for a time frame of $0-5$ it means that the version with a timestep of 0.5 has to do twice the amount of steps as the version with 1. However, the version with a timestep of 1 might not accept the required tolerance even when calculating the timestep in smaller parts. This means that the Bulirsch-Stoer step will reset and will do the calculations for a timestep of 0.5 anyway and the time it took trying to get accepted with a timestep of 1 for the set tolerance is lost.

Table 1: Overview of the timings for the initial conditions generated according to the Plummer model in milliseconds with a tolerance of 1e$-$2 and 64 bits precision.

Generation	100 (CPU)	100 (GPU)	300 (CPU)	300 (GPU)	1000 (CPU)	1000 (GPU)
0	102238	79432	424986	173229	2647468	2681570
1	57039	97747	914470	460682	1178226	957392
2	83730	334087	128234	212052	618980	692843
3	48010	56840	72983	87205	1182511	1357074
4	95708	298640	729737	275512	10003805	2721348
5	218940	268995	624530	402219	1700345	712296
6	166034	147390	493011	233693	3458685	1147785
7	783274	1734221	1076701	233120	17666392	2305370
8	38952	100783	648429	252590	434931	250018
9	23295	98753	449034	313792	514582	474402
Average	161722	321689	556212	264409	3940593	1330010

6.3 Optimization Timings

In this section we elaborate on the timings of the different optimizations which we described in detail in section 4.6. The setup of these timings can be found in section 5.3. In between the version with optimized scratch space and the version with a separate meta-data there is another version, which we did not mention yet in the optimizations. This version has a partial implementation of the last optimization with a stepsize for the operations to coalesce the global memory accesses. Later we optimized this further into the last optimization and therefore it is not described as a separate optimization. However, we will also show the timing of this partially optimized version to show the difference the optimization of the separate meta-data made.

Table 2 shows the overview of the timings to calculate one timestep for different amount of bodies and different amount of bits arbitrary precision when using the different optimizations described in section 4.6. The choices for the amounts of bodies and bits arbitrary precision that are used are explained in more detail in section 5.3. If the calculation of one timestep with infinite Bulirsch-Stoer tolerance is irrelevantly long or simply not possible because of the limited amount of shared memory per streaming multiprocessor of the GPU a "-" is used.

Table 2 shows that our first version with only an optimized extrapolation is more than two times slower for 100 bodies and 64 bits arbitrary precision than our fully optimized version is for ten times that amount of bodies and the same amount of bits arbitrary precision. Table 2 also shows that using shared memory look very promising at first, but could not be scaled to the point of 1000 bodies using 512 bits arbitrary precision. Also using shared memory limits the occupancy of the GPU as we already mentioned in section 4.6.3. The difference between the partial optimized version and the shared memory version shows that using partially optimized global memory already is faster than using shared memory because of the higher possible occupancy of the GPU when using global memory. This partially optimized global memory is further optimized by separating the meta-data from the limbs (section 4.6.4) and re-aligning all when performing the arbitrary precision arithmetic (section 4.6.5). These last two optimizations mostly show an increased performance when the amount of total used global memory grows, which was expected because memory accesses become a lot less efficient when there is more memory in between the different accesses that are done at the same time.

Table 2: Overview of the timings for the different optimizations for one timestep in milliseconds.

Optimization	100 (64-bit)	100 (512-bit)	1000 (64-bit)	1000 (512-bit)
Extrapolation	5475	-	-	-
Scratch	1676	306192	-	-
Shared memory	74	6110	11029	-
Partial optimization	36	516	4079	371595
Separate meta-data	36	488	2600	166606
Stepsize for operations	30	471	2215	32875

7 Conclusions

This thesis presented a first version of N-body program Brutus using a GPU. It explained the main concepts of GPU programming and the related optimizations that were used to speed up the program when using a GPU. In this chapter we will conclude by answering the problem statement and research questions we addressed in the introduction.

7.1 Brutus on GPU

The possibility of an efficient implementation of Brutus[7] on a GPU by parallelizing over the amount of bodies was researched. The results showed that the speedup of the implemented and optimized GPU version depends on the number of bodies and the amount of bits arbitrary precision. A speedup of up to 20 times for the lowest precision of 64 bits was shown. This speedup becomes less when scaling the amount of bits arbitrary precision. For 100 bodies the GPU implementation becomes slower than the CPU implementation when using the highest amount of bits arbitrary precision. For 300 bodies the GPU implementation is still faster than the CPU implementation, but only 1.4 times faster for the highest amount of bits arbitrary precision. Finally for 1000 bodies the speedup of the parallel version is still 5 times when using the highest amount of 512 bits arbitrary precision.

7.2 Memory Usage

Our first research question stated: *How to efficiently use GPU memory and deal with its limitations when using arbitrary precision on a GPU?*. The results in Table 2 show that using shared memory combined with using arbitrary precision results in low performance because using a lot of shared memory results in a low occupancy of the GPU. Table 2 also shows that with growing amounts of data, coalescing global memory accesses of data is important for performance. It shows that the optimizations that were done to make memory accesses coalesced, improved the performance of the program to the point where there was speedup for initial conditions with a large number of bodies.

7.3 CPU/GPU Overhead

Our second research question stated: *How to minimize CPU/GPU overhead?*. The problem of CPU/GPU overhead with large amounts of data has been overcome by keeping all the data that is necessary for the calculations on the GPU. This means that a few calculations that are hard to parallelize are performed slower than they would be on CPU, but these few calculations are a small part of the total calculation time and therefore can be ignored compared to the CPU/GPU overhead that it would have cost to run these calculations using the CPU.

7.4 Generic code

Our third research question stated: *How to make the code as generic as possible with respect to the GPU architecture?*. The presented GPU program has been made while keeping different GPU architectures in mind. It also compiles using CUDA with compute capabilities 2.0 and higher. There has been taken into account that different CUDA compute capabilities require different values for the kernel calls. Although there has not been provided an optimal configuration for all compute capabilities, the related parameters are stored in global variables and are adjustable to the optimal values for the used compute capability.

7.5 Summary

In the end an optimized GPU implementation of N-body program Brutus was provided with an increased performance compared to the current optimized CPU version when scaling the amount of bodies. The speedup depends on the number of bodies and the amount of bits arbitrary precision that are used. Furthermore a run for a semi-realistic initial condition and configuration were shown as well as the influence of the optimizations.

8 Discussion and Future Work

In this chapter we will briefly discuss the usefulness of our work in the area of astronomy and provide possible future work, which we think will increase the performance even further.

8.1 Discussion

As was shown in Figure 10, the optimized GPU version of Brutus that was described in this thesis is not faster than the optimized CPU version for an amount of bodies lower than 40 and therefore it will not be useful to astronomers for smaller N-body problems. On the other hand we have shown a significant improvement for calculating with an amount of bodies higher than 300, even when the amount of bits arbitrary precision is growing. This makes our optimized GPU version useful for larger N-body problems as long as the number of bodies is large enough compared to the amount of bits arbitrary precision that is used. For a large number of bodies and a small amount of bits precision our optimized GPU version is always faster than the current CPU version. For a small number of bodies and a large amount of bits precision our optimized GPU version is never faster than the current CPU version. For a large number of bodies and a large amount of bits precision our optimized GPU version is still faster, but not as much as with a smaller amount of bits precision. Finally it is interesting to keep in mind that currently the speed of GPUs is growing faster than the speed of CPUs, so the difference in speed will change in favor of the GPU version over time without any further optimizations.

8.2 Future Work

For possible further speed up we planned to parallelize the Bulirsch-Stoer algorithm over multiple devices. S. Sultan, who we already mentioned multiple times in this thesis, has already tested a CPU version which uses multiple CPUs. The results were promising and therefore it seem like the same parallelization would help for our GPU version as well. This could be implemented by using multiple GPUs connected with NVLink [6]. Data can be transferred directly from one GPU to another GPU without having to pass the CPU by using GPUDirect [5].

References

[1] CPU vs GPU. http://blog.goldenhelix.com/wp-content/uploads/2010/10/cpu_vs_gpu.png. 2016 (Accessed December 9, 2016).

[2] The GNU multiple precision arithmetic library. https://gmplib.org/. 2017 (Accessed January 20, 2017).

[3] Modified Midpoint Method XMDS2 2.2.3 documentation. http://www.xmds.org/_images/richardsonExtrapolation.png. 2017 (Accessed January 9, 2017).

[4] Netherlands' smallest supercomputer. https://www.universiteitleiden.nl/en/news/2017/04/dutch-smallest-supercomputer. 2017 (Accessed December 15, 2017).

[5] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect. 2017 (Accessed December 22, 2017).

[6] NVIDIA NVLink High-Speed Interconnect. http://www.nvidia.com/object/nvlink.html. 2017 (Accessed December 21, 2017).

[7] Nvidia profiler. https://developer.nvidia.com/nvidia-visual-profiler. 2017 (Accessed December 22, 2017).

[8] T. Boekholt and S. Portegies Zwart. On the reliability of n-body simulations. *Computational Astrophysics and Cosmology - Springer Journals*, March 2015.

[9] Kevin J. Bowers, Ron O. Dror, and David E. Shaw. The midpoint method for parallelization of particle simulations. *The Journal of Chemical Physics*, 124(18):184109, 2006.

[10] Clarence Burg and Taylor Erwin. Application of richardson extrapolation to the numerical solution of partial differential equations. *Numerical Methods for Partial Differential Equations*, 25(4):810–832, 2009.

[11] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[12] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 47–47, Nov 2004.

[13] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[14] Khronos OpenCL Working Group et al. The opencl specification. *A. Munshi, Ed*, 2008.

[15] Pawan Harish and P. J. Narayanan. *Accelerating Large Graph Algorithms on the GPU Using CUDA*, pages 197–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[16] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.

[17] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically managed data for cpu-gpu architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165–174, New York, NY, USA, 2012. ACM.

[18] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[19] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. May 2010. https://arxiv.org/abs/1005.2581.

[20] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. m. Hwu. Gpu clusters for high-performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, Aug 2009.

[21] Ronald T. Kneusel. *Arbitrary Precision Floating-Point*, pages 265–292. Springer International Publishing, Cham, 2017.

[22] Bernhard Langer. Arbitrary-Precision Arithmetics on the GPU. 2015. http://old.cescg.org/CESCG-2015/papers/Langer-Arbitrary-Precision_Arithmetics_on_the_GPU.pdf.

[23] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[24] Kenneth Meyer, Glen hall, and Daniel C. Offin. Introduction to hamiltonian dynamical systems and the n-body problem. pages 27–35. Springer-Verlag, 2009.

[25] Kenneth R. Meyer, Glen R. Hall, and Dan Offin. *Introduction to Hamiltonian Dynamical Systems and the N-Body Problem.* Applied Mathematical Sciences Volume 90. Springer, New York, 2008.

[26] T. Nakayama and D. Takahashi. Implementation of multiple-precision floating-point arithmetic library for gpu computing. *Proc. 23rd IASTED International Conference on Parallel and Distributed Computing and Systems*, 2011.

[27] John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[28] NVIDIA. Cuda c programming guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. 2016 (Accessed November 21, 2016).

[29] NVIDIA. Streaming multiprocessor (SM). https://devblogs.nvidia.com/wp-content/uploads/2014/09/GeForce_GTX_980_SM_Diagram-545x1024.png. 2016 (Accessed December 12, 2016).

[30] Gregory D. Peterson, Akila Gothandaraman, Rick Weber, and Robert J. Hinde. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel & Distributed Systems*, 22(undefined):58–68, 2010.

[31] H. C. Plummer. On the problem of distribution in globular star clusters. 71:460–470, March 1911. http://adsabs.harvard.edu/abs/1911MNRAS..71..460P.

[32] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[33] sfackler, jcmacdon, FooManchu, and Arnie. Cuda device memory model. http://15418.courses.cs.cmu.edu/spring2013/article/11. 2016 (Accessed December 12, 2016).

[34] Josef Stoer, Roland Bulirsch, Richard H. Bartels, Walter Gautschi, and Christoph Witzgall. *Introduction to numerical analysis*. Texts in applied mathematics. Springer, New York, 2002.

[35] Shlomo Weiss and James E. Smith. *IBM Power and PowerPC*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[36] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. *SIGPLAN Not.*, 48(8):57–68, February 2013.