High Performance Direct Gravitational N-body Simulations on Graphics Processing Units

An implementation in CUDA

MSc Thesis (*Afstudeerscriptie*)

written by

Jeroen Bédorf

under the supervision of **Dr. R.G. Belleman** and **Dr. S.F. Portegies Zwart**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Grid Computing

at the *Universiteit van Amsterdam*.

Date of the public defence:
November 16, 2007

Members of the Thesis Committee:
Dr. P. van Emde Boas
Dr. A. Ponse
Dr. R.G. Belleman
Dr. S.F. Portegies Zwart

UNIVERSITEIT VAN AMSTERDAM

Abstract

At the end of 2006 NVIDIA introduced a new generation of graphical processing units (GPUs) (the so called G80 architecture). These GPUs are more powerful than any of the GPUs released before; they offer up to 350 billion floating-point operations per second (GFLOP/s) in certain situations. With the introduction of this hardware NVIDIA released a new programming environment that makes it easier for programmers to use the GPU for other tasks than graphics. This software environment is called Compute Unified Device Architecture (CUDA) and interacts closely with the hardware characteristics of the G80 architecture.

In this thesis CUDA is used to implement an N-body algorithm that is suitable for astrophysical direct N-body integrations. The performance of the GPU will be compared with the GRAPE-6Af special hardware. To compare the GPU and the GRAPE we have implemented our algorithm in a library that is compatible with the GRAPE library. This library is linked with the `starlab` software package to compare performance and relative error between the GPU and the GRAPE. Results show that the GPU can match the performance of the GRAPE and achieve a comparable error when a relative large energy error is acceptable. For high precision simulations the GRAPE outperforms the GPU, because the double precision on the GRAPE hardware allows the algorithm to take larger time steps while maintaining a higher accuracy. The GPU hardware is single precision, but double precision GPUs have been announced in various NVIDIA documents.

The GPU code can easily be adapted for other N-body algorithms by changing the equations that are used. This enables researchers from other research areas to speed up their codes by several orders of magnitude at a relative low cost.

Acknowledgements

I would like to thank my supervisors Robert Belleman and Simon Portegies Zwart for their advice and suggestions. I am grateful to Mark Harris and David Luebke of NVIDIA for supplying us with the two NVIDIA GeForce 8800GTX graphics cards on which the simulations were performed. I would like to thank Jorrit Adriaanse from SARA for rebooting and upgrading the computers when the CUDA environment had put the machine in a non responding state. The calculations for this work were done on the Hewlett-Packard xw8200 workstation cluster and the MoDeStA computer in Amsterdam; both are hosted by SARA Computing and Networking Services, Amsterdam.

Contents

Chapter 1

Introduction

The introduction of multiple processing cores in one chip allows microprocessor manufacturers to improve the performance of CPUs while the clock rate stays the same. This multi-core principle is not new. Over the last two decades, a similar approach has been taken by manufacturers of graphics processing units (GPU) under the influence of the gaming industry to deliver increasingly detailed and responsive computer games. As a result of this, the GPU underwent a dramatic increase in performance; a doubling in performance over a period of 9 months, instead of 18 months for CPUs [6, 22].

In terms of raw performance, today's GPUs offer more performance than conventional CPUs. For example, the latest GPU from NVIDIA, the G80 mounted on a GeForce 8800GTX, has a performance of about 350 GFLOP/s (see chapter 4). However, harvesting this computing power is not trivial as GPUs are designed and optimized for graphics operations. Over the last 7 years GPUs have evolved from fixed function hardware for the support of primitive graphical operations to programmable processors that outperform conventional CPUs, in particular for vectorizable parallel operations. Today's GPUs contain many smaller processors called stream processors [28], which are specialized in processing large amounts of data in a streaming and parallel fashion. It is because of these developments, that more and more people use the GPU for wider purposes than just for graphics [8, 30, 5]. This type of programming is also called general purpose computing on graphics processing units (GPGPU[1]).

Initially, the programming of GPUs was done in assembly language and required a very specific knowledge of the hardware. Newer generations of GPUs offered more possibilities for the programmer and with this came the need for high-level programming languages. With the introduction of shading languages like Cg [20] and GLSL [14], the programmer could focus on the problem at hand.

Around this time, the performance of the GPU attracted the attention of re-

[1] See also `http://www.gpgpu.org`

searchers with an interest in the use of the GPU as a high-performance coprocessor. First implementations mapped their problems into a graphics problem where data is represented as coloured pixels stored in textures. Shading programs were then used to perform computations on the data. Although not every problem is easily represented as a graphics problem, the use of the GPU was demonstrated in many scientific areas, including but not limiting to PDE solvers, ray tracing, image segmentation and gravitational simulations [27, 29, 30, 32].

To make GPUs more suitable for general purpose computing, NVIDIA released at the end of 2006 the Compute Unified Device Architecture (CUDA)[6]. This is a programming framework that consists of a special compiler that compiles "C" into code that can be executed by the GPU [2]. The CUDA architecture is tightly connected to the GPUs hardware architecture for which it is designed: the G80 architecture released around the same time. The combination of hard- and software makes it easier for the programmer to design programs that are suitable to run on GPUs. In this thesis we make use of the CUDA framework.

This thesis is based on the paper that is written about this work [3].

1.1 Thesis overview

This thesis describes the implementation of an N-body simulation code on a GPU using CUDA.
Chapter 2 describes the architecture of the G80 GPU and the programming model of CUDA. CUDA is tailored around the G80 hardware architecture and the chapter explains how these two are related.
In chapter 3 the N-body problem is explained and two different simulation methods are shown (direct integration and treecode). After the details of the N-body problem are explained, we will show how the N-body problem is mapped onto the GPU architecture.
Chapters (4 and 5) cover the results and the discussion of these results, the performance of various simulations has been compared for the GPU, CPU and the GRAPE. The simulations are performed using various algorithms and accuracy settings. In addition to the simulation results also the performance of the architecture is presented. The last sections of chapter 5 show the scaling when using multiple GPUs and cover upcoming changes that will be made to the GPU and the CUDA programming environment.
Finally, Chapter 6 contains the conclusion.

[2] The "C" code has to be written especially for the GPU, it is not possible to just convert any existing program.

Chapter 2

Background

2.1 Introduction

The use of the GPU for general purpose computing is relatively new. Traditionally computational tasks where executed on the CPU and if it would prove to be too much work for one CPU then the work was divided over multiple CPUs. The last few year's researchers have started using the GPU as an extra computational unit on which parts of the algorithm are run to get performance improvements over using only the CPU. The GPU offers more computational power, in terms of the number of floating point operations that can be performed per second (typically this is depicted in "GLOP/s") than a single CPU can, because of the different design of the GPU. The GPU is a so called stream processor; the next section explains the difference between the CPU and the GPU.

2.2 The Stream Processor

The CPU found in PCs is a processor that is designed for general purpose computing. It is a complex processor that consists for a large part out of control hardware. This is used for e.g. re-ordering of instructions, branch prediction, prefetching, interrupts and task switches. Only a small part of the transistors that make up a CPU is actually used for computational purposes. Traditionally, the CPU is single threaded and consists out of one core. Since the processor is single threaded it is not build for parallel computations. Even though the CPU has pipelining mechanisms and context switches the basic design is single threaded. With the latest generation of processors a change in this single threaded principle takes place with the development of processors that contain multiple cores (2 and 4 at the moment of writing). These changes to the CPU

increase the performance for multi-threaded applications, but the part of the processor that is dedicated to control logic is still very large. This is caused by the way the multi-cores are designed: on the core level it consists out of two separate processors each with its own control logic. There is even more control logic needed to let the two (or more) cores operate together.

The Graphics Processing Unit (GPU) found on modern graphic cards has a different design and is not designed to be a general purpose processor, but as a processor that needs to process millions of data elements (polygons, pixels, colours, etc.) in a limited amount of time. The GPU is a so called "Stream Processor" [28] and is designed for graphical rendering of images. In a stream processor the data "streams" through various processing parts of the processor. These parts are called kernels. The kernels operate on data streams. The kernels are implemented in hardware which greatly improves the performance compared to a software implementation on a general purpose processor. In the first generation of GPUs the kernels could only perform build-in tasks like transformations, rasterization, lighting and culling, but with the newer generations two of the kernels can be programmed by the user. These two kernels are called the vertex and fragment kernels (sometimes also referred to as shaders). They are used by the programmers to either implement 3D effects like shadows, reflections, etc. or to implement general purpose algorithms.

In 2006 Microsoft introduced Direct3D 10 [4] as a new standard for graphics APIs. In order to make their GPUs Direct3D 10 compatible, producers had to add another programmable kernel to the card. This kernel is called the "geometry shader" and is used to generate new geometry on the GPU itself. The two biggest GPU producing companies (ATI and NVIDIA) have both changed the way they implement the programmable kernels in hardware. Instead of creating specific kernels for the vertex, fragment and geometry shaders they have implemented universal kernels that can be used for all three shaders. This results in an overall increase of the total number of kernels that can be used by the programmers (graphics programmers and GPGPU programmers alike).

To utilize the full power of a stream processor there has to be enough data that can stream through the various kernels of the processor. The data is then processed by one kernel after each other with minimal memory latencies. Most stream processors are focused on particular tasks and can not be efficiently used for any other tasks which make them unsuitable for applications that are not parallel and contain many different code paths caused by branches. They can however be used to implement part of an algorithm that requires high computational power. Implementing algorithms on a GPU can be done in a variety of programming languages. These languages can be distinguished in two types. One is called "graphics languages" and is used for graphical effects. The other one is called "streaming languages" focused on general purpose computations.

2.3 Programming Languages

When programming a GPU a choice exists out of two types of programming languages, namely "graphics" languages and "streaming" languages. These languages try to make it easier for the programmer to create the effects and results he or she wants to achieve without having to program in assembly language.

Graphics Languages These languages are used to create complex graphical effects (shadows, illumination, bump mapping, etc.). This requires the programmer to learn a new programming language with all the specific language rules and variable types. These languages are also called "shader languages" since they are used to program the shader kernels on the GPU.

An overview of the various graphics languages can be found in [10].

Streaming Languages These languages more closely resemble existing programming languages and are focused on general purpose computation rather than the creation of graphical effects. They are implemented as extensions to languages like "C" or as additional libraries that make it easier to create parallel programs and complex data types like matrixes and vectors.

The choice of language depends on the experience of the user, his needs and most of all what type of application they are planning to create. When creating a graphical engine the obvious choice is to use one of the graphical languages for the tight interaction with the hardware architecture and the interoperability with graphics APIs. On the other hand when the focus of the program is general purpose both types of language can be used. Since both types generally require some prior knowledge about the language the choice usually is related to the programmer's experience. The last thing that has to be taken into account is the platform on which the program has to run. Not all languages can run on Linux and Windows and not all of the streaming languages offer support for both NVIDIA and ATI GPUs.

In this thesis we make use of NVIDIA "Compute Unified Device Architecture" (CUDA), since it was expected that CUDA could offer more performance than any previous language [10]. Besides that we have two NVIDIA GPUs available that were the fastest GPUs available at the moment that we started this project (and they still are among the fastest cards available) and only CUDA is able to access all the different types of memory present on these GPUs.

2.4 The CUDA architecture

2.4.1 Introduction

The "Compute Unified Device Architecture" (CUDA) [7] was developed by NVIDIA to give applications access to the GPU through an easy to use programming interface. CUDA is a combination of hard- and software and will only work on specific GPUs made by NVIDIA. For the hardware part a CUDA capable GPU[1] is required to be able to make use of the programming interface. The CUDA programming environment is based on the "C" programming language which is extended with a number of keywords and data types that make it easier to create programs that use the GPU and its complicated memory model (see §2.4.2). The software part consists of a special graphics card driver and the CUDA toolkit that contains the libraries and tools that are necessary to build applications. The CUDA framework comes with two mathematical libraries that are optimized for the GPU, namely BLAS and FFT. CUDA has been designed to be scalable for newer generations of NVIDIA GPUs which makes it attractive for programmers since it will not be abandoned when a new generation of GPUs is introduced. The downside of the CUDA package is that it only works on NVIDIA cards since it is developed for NVIDIA hardware.

2.4.2 Hardware - G80 architecture

The CUDA architecture is tightly connected to the G80 hardware architecture. The programming interface gives access to different parts of the GPU and makes it possible to start computations. The next paragraphs cover, in detail, the different aspects of the G80 hardware; Processors, Memory, Bundles and Threads.

Processor lay-out The graphic card used for this project is a GeForce 8800GTX and contains 16 multiprocessors[2]. A multiprocessor is composed of eight processing units (PU) each. This adds up to a total of 128 processing units on the GeForce 8800GTX. Besides the processing units a multiprocessor contains different types of memory, namely registers, caches (texture and constant) and shared memory. Slower types of the G80 contain fewer multiprocessors. Future types are expected to contain more multiprocessors which increase the number of total processing units available. The processors on the multiprocessor operate using a "Single Instruction Multiple Data" (SIMD) architecture. This implies that each processor on the multiprocessor executes the same instruction at the same moment. This has the implication that the alternatives in a conditional execution path (commonly referred to as "a branch") cannot be executed by all PUs at the same time, i.e.: first the PUs that take one side of the branch will

[1] At the moment of writing the G80 and G84 architectures are supported by CUDA.

[2] The G80 used on the GeForce 8800GTX is the fastest CUDA capable GPU at the moment of writing.

execute while the other PUs are stalled, then the PUs that take the other side of the branch will execute while the first PUs are stalled. Concurrent execution continues after all PUs have returned at the same point in the code.

The processing units are fully capable processors that run at twice the clock speed of the rest of the card and are capable of executing complex math functions (see the CUDA documentation [26] for a table of all supported functions).

Current GPUs support 32-bit IEEE floating point numbers, which is below the average general purpose processor, but for many applications it turns out that the higher (double) precision can be emulated at some cost or is not crucial. The G80 follows the IEEE-754 standard for single-precision binary floating-point arithmetic. However it does not support all points of the definition. The list of deviations can be found in Appendix A.

Figure 2.1 shows an overview of the processor layout. The hardware architecture is reflected in the software interface provided by CUDA. CUDA defines the following concepts to access the hardware architecture

To start computations on the GPU, using CUDA, the number of threads that have to be launched needs to be specified. A thread is the program code that is executed by the GPU. Threads are contained in bundles[3]. Each bundle contains the same amount of threads and a group of bundles is called a grid. To start execution, the number of bundles and the number of threads per bundle that is started need to be specified. The execution order of the bundles is undefined with the result that communication / synchronization between bundles is not possible. However communication / synchronization between threads that are contained in the same bundle are possible by making use of shared memory (see next paragraph). Communication between threads is introduced by the CUDA architecture (it is not possible to communicate between threads when for example Cg or GLSL is used to operate the GPU).

The NVIDIA G80 hardware architecture defines a hierarchical memory structure where each level has a different size, access restrictions and access speed. In general, accessing the largest type of memory is flexible but slow, while accessing the smallest type of memory is restrictive but fast. This memory structure is directly exposed by the CUDA programming framework. The challenge in mapping a computing problem efficiently on a GPU through CUDA is to store frequently used data items in the fastest memory, while keeping as much of the data on the device.

The following memory types can be distinguished:

- Global memory, this is the largest type of memory and is off-chip memory that is not cached. This memory is the slowest of the different memory types (access time between 400 and 600 ms).

[3]Instead of “bundle” NVIDIA uses the term “block”. We have chosen “bundle” to avoid confusion with another meaning of “block” later in this work.

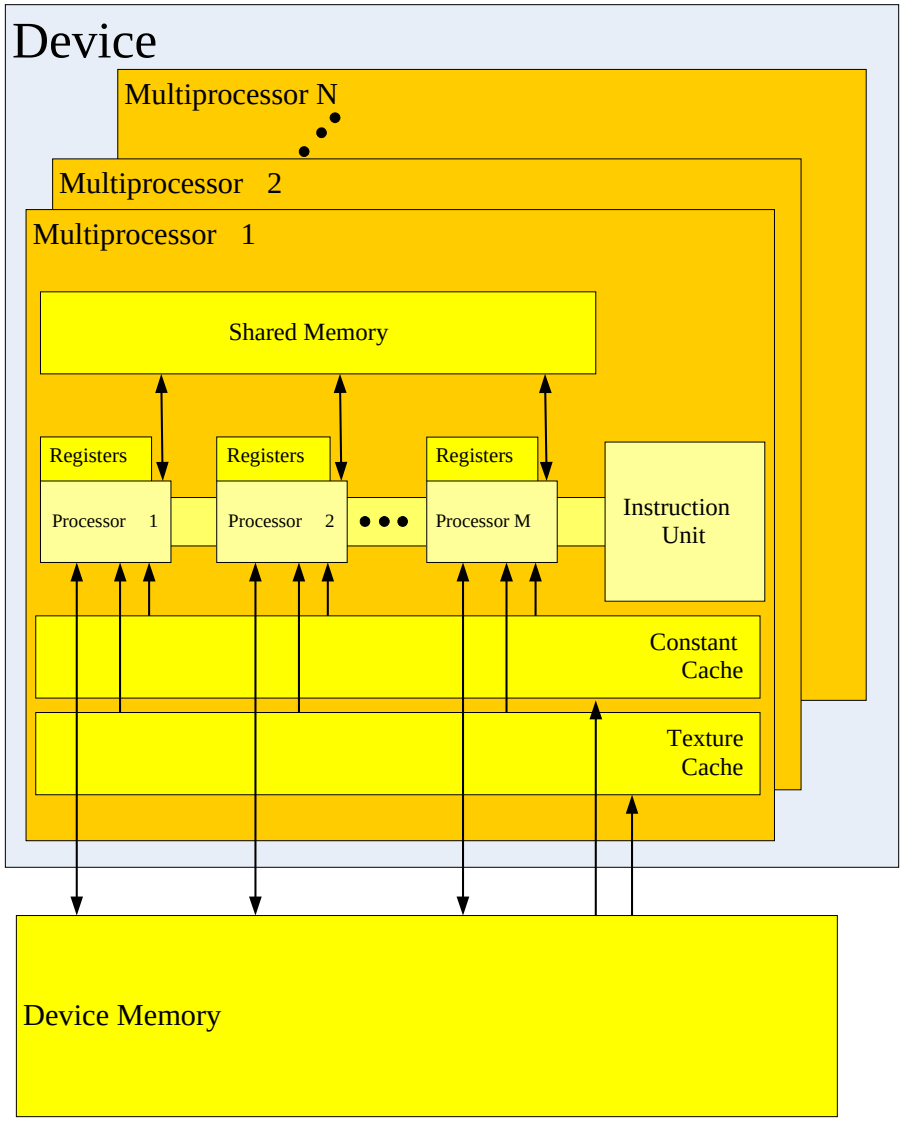

Figure 2.1: Graphical representation of the G80 architecture. The G80 that we use has N=16 multiprocessor with M=8 processors each. Image taken from [25].

- Constant and texture memory are placed in the same memory space as the global memory, but they are cached. The caches are placed on the multiprocessor which makes access to them fast and with low latency. However both the constant and texture memory are read-only memory types.
- Shared memory, this memory is limited in size and is placed on the multiprocessor itself. It is accessible by the bundles that are executed on the multiprocessor. This memory is very fast (access time around 2 millisecond).
- Registers, this is the fastest type of memory and is placed on the multiprocessor. Registers are thread specific and cannot be shared between threads.

All these memory types can be accessed directly through CUDA (with the exception of the registers that are handled by the compiler). The trick is to make optimal use of the different types of the memory to reduce access to the slower memory types. This can be done by copying data from global memory into shared memory or by making use of the texture and constant memory caches. Data that resides in global, constant or texture memory can be copied to the host; data that resides in the registers or in shared memory has to be moved to the global memory space before it can be accessed from the host. To prevent that data in shared memory gets invalid, a bundle is always bound to the same multiprocessor on which it is started and gets assigned a specific part of the shared memory space.

Figure 2.2 shows the different memory types and their relation to each other and their relation to the bundles and threads.

2.4.3 Software

Driver structure The graphics cards come with a special driver that handles the interaction between the OS and the graphics card. This driver contains functionality for displaying 2D and 3D graphics as well as support for the various graphics APIs like OpenGL and Direct3D. This poses a substantial overhead when the GPU is only used for computations that do not require that functionality. Therefore CUDA uses a separate driver than the driver used for displaying graphics. The CUDA driver is optimized for general purpose computations and contains less overhead than the display driver. On top of the driver there is the CUDA runtime that is accessible by applications to interact with the GPU. To get more control over the application it is possible to interact with the driver directly. For normal use however it is easier to use the runtime library. The memory transfers that are issued by the driver make use of Direct Memory Access (DMA). DMA transfers are faster than normal memory transfers and are less CPU intensive, because the data is copied from one memory type to the other without intervention of the CPU.

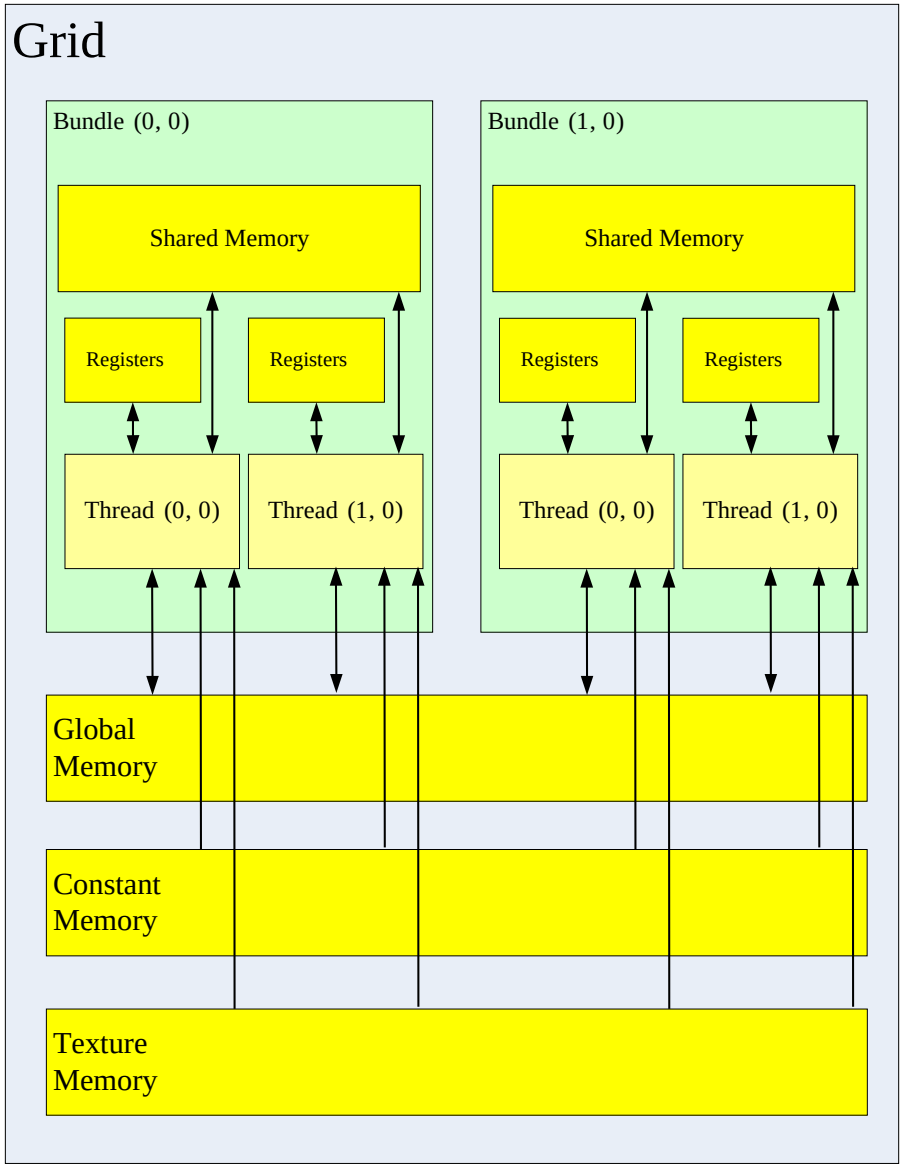

Figure 2.2: Graphical representation of the memory architecture of the G80 GPU. Image based on images from [25].

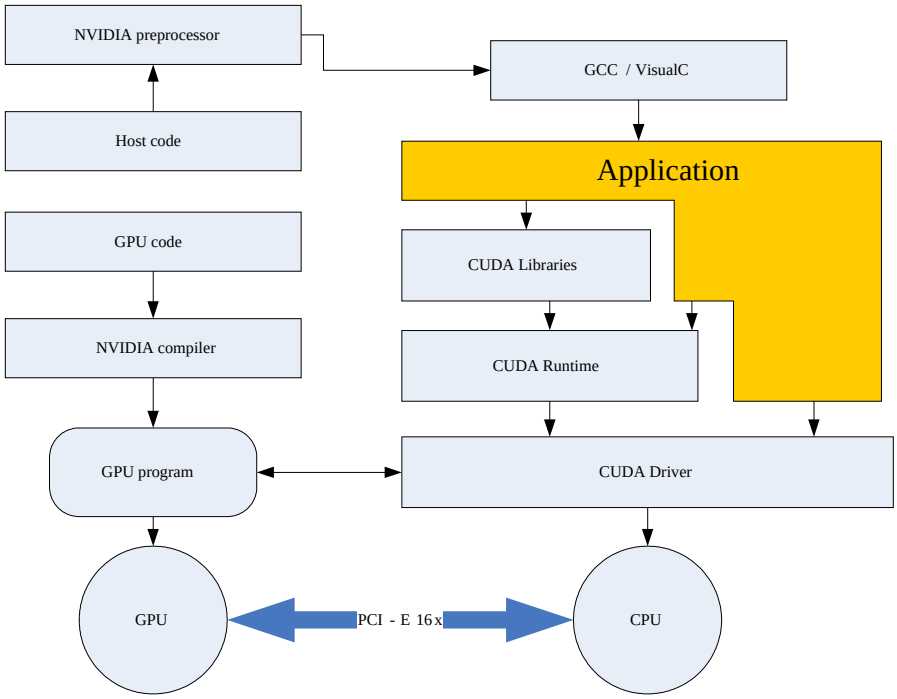

Figure 2.3: Graphical representation of the programming model in relation to the driver and compiler architecture. Image based on images from [25].

Programming model A CUDA program consists of two parts: the host code and the device code.

The host code is run on the CPU. In this part the memory is initialized, memory copies are executed, computations are started and all in- and output is handled. The device (GPU) code is the part that runs on the GPU. This part handles all executions on the device, the required calculations, storing of results, synchronization, etc. It is not possible to write host code in the device code, e.g. functions like "printf" can not be executed on the GPU.

Compiler The CUDA environment comes with a special compiler that compiles the host and GPU code. The GPU code is compiled into machine language and the host code is converted to "C" code that can be processed by the usual compilers (Visual C, g++, etc.). The host code has to be processed since CUDA makes use of a number of non-standard keywords that will be converted to ANSI C-code.

Figure 2.3 shows the relation between the different parts of the source code and various run time libraries.

Chapter 3

N-Body

3.1 N-body problem

The N-body gravitational algorithm is based on the force equation as discovered by Newton. The equation calculates the force between two particles in space:

$$\mathbf{F}_i \equiv m_i \mathbf{a}_i = m_i G \sum_{j=1, j\neq i}^{N} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}. \tag{3.1}$$

Here G is the Newton constant, m_i is the mass of star i and $\mathbf{r}_i$ is the position of star i. The total force $\mathbf{F}_i$ (or the acceleration $\mathbf{a}_i$) that is exercised on particle i is the summation over the forces between i and all N particles.

In order to determine the total force on each particle within an N-body system, the force exerted by all N particles has to be calculated. Calculating the force of all particles in the N-body system requires $\frac{1}{2}N(N-1)$ force calculations. This $\mathrm{O}(N^2)$ part of the algorithm is the computationally most expensive part. The calculation of the force exerted on each particle is independent of the calculations performed for other particles. This makes the calculation of the forces for all particles parallelizable. GPUs contain many processing units that each perform the same series of similar operations on different streams of input data, a technique which is better known as Single Instruction Multiple Data (SIMD). It is this part of the algorithm that is ported to the GPU.

3.1.1 Direct Integration

The most commonly used integration scheme today is the predictor-corrector scheme in combination with a fourth order integrator. This scheme was first introduced by Makino & Aarseth in 1992 [18]. The scheme uses individual

time steps which greatly improves accuracy over previously used integration methods that used shared time steps. This scheme was extended to support block time-steps in 1991 by Makino [15] and further refined in 1993 by McMillan and Aarseth [21]. The block time-step extension made it possible to efficiently execute the algorithm on special purpose hardware like the GRAvity PipE (GRAPE, see § 3.2 for more details). In a block-time step algorithm particles are integrated in blocks of particles. Particles in the same block are all integrated at the same moment (in parallel if the hardware allows that). The block sizes vary between the range 1 to N. If all the blocks are of size 1 then the algorithm is executed as an individual time-step algorithm. When all blocks contain N particles then a shared time-step model is executed. The block-size will be represented by n in the following chapters.

The predictor-corrector scheme with block time-step goes through the following steps in each time step :

1. Determine which particles have to be updated and place them in the same block.
2. Predict the position and velocity of all particles. This can be done on specialized hardware if it is available.
3. Integrate the particles in the current block. For each particle we determine the force, jerk and potential. This can be done on specialized hardware if it is available.
4. Correct the integrated particles and save the new position and velocity and determine the next time that the particles have to be integrated.
5. Go back to step 1 until we have simulated the requested period.

Details

To allow grouping of particles, the time steps are taken as powers of 2. This way, the chance that particles have the same time-step is much higher than when all particles have a time-step that is not based on a power of 2, since time steps that would have been the same now can differ in the lowest bit. Each particle has two values associated with it: the time of the last update t_{last} and the time-step after which it has to be updated again Δt_j. To determine the particles that have to be updated, we first loop through the particles to find the smallest next time step. When we have found the smallest time step we select all the particles for which $(\Delta t_j + t_{last})$ is equal to the smallest time step.

When the particles are selected, the position and velocity for the next time step $(\Delta t_j + t_{last})$ is predicted for all N particles. When the GRAPE hardware is available, the prediction can be done on the GRAPE hardware. Equation 3.2 shows the prediction equation.

$$\mathbf{x}_{p,j} = \mathbf{x}_j + (t - t_j) + \frac{(t - t_j)^2}{2}\mathbf{a}_j + \frac{(t - t_j)^3}{6}\dot{\mathbf{a}}_j.$$
and
$$\mathbf{v}_{p,j} = \mathbf{v}_j + (t - t_j)\mathbf{a}_j + \frac{(t - t_j)^2}{2}\dot{\mathbf{a}}_j. \tag{3.2}$$

Here $t - t_j$ is the time-step for particle j, $\mathbf{a}$ is the acceleration and $\dot{\mathbf{a}}$ is the derivative of the acceleration.

After the prediction is completed the integrator can be executed to determine the acceleration $\mathbf{a}$, jerk (derivative of the acceleration) $\dot{\mathbf{a}}$ and the potential ϕ for each particle in the block. This is the computational most intensive part of the algorithm and has a computational complexity of $O(N^2)$. These calculations can be executed on specialized hardware (like the GRAPE or the GPU) to obtain a performance gain over the CPU.

Equations 3.3, 3.4 and 3.5 show the required calculations for the integrator [24].

$$\mathbf{a}_i = \sum_{j=1, j\neq i}^{N} \frac{m_j \mathbf{r}_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}}. \tag{3.3}$$

$$\mathbf{j}_i = \dot{\mathbf{a}}_i = \sum_{j=1, j\neq i}^{N} m_j \left[\frac{\mathbf{v}_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}} - \frac{3(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})\mathbf{r}_{ij}}{(r_{ij}^2 + \varepsilon^2)^{5/2}} \right]. \tag{3.4}$$

$$\phi_i = - \sum_{j=1, j\neq i}^{N} \frac{m_j}{(r_{ij}^2 + \varepsilon^2)^{1/2}}. \tag{3.5}$$

Where $\mathbf{r}_i$ stands for the position, $\mathbf{v}_i$ for the velocity and m_i for the mass of particle i, $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ and $\mathbf{v}_{ij} = \mathbf{v}_j - \mathbf{v}_i$. Gravity is taken as unity and is therefore ignored in the equations.

The last step in the algorithm is correction of the predicted particles. The position and velocity of particles that are integrated in the previous time step are corrected using the newly calculated force and jerk. This is done using a third-order Hermite interpolation polynomial [18] as expressed in equations 3.6 and 3.7.

$$\mathbf{a}_i(t) = \mathbf{a}_{0,i} + \Delta t \dot{\mathbf{a}}_{0,i} + \frac{\Delta t^2}{2}\mathbf{a}_{0,i}^{(2)} + \frac{\Delta t^3}{6}\mathbf{a}_{0,i}^{(3)}. \tag{3.6}$$

Here $\Delta t = t - t_i$, $\mathbf{a}_0$ and $\dot{\mathbf{a}}_0$ are the acceleration and its time derivative calculated at time t_i; $\mathbf{a}_{0,i}^{(2)}$ and $\mathbf{a}_{0,i}^{(3)}$ are given by

$$\begin{aligned} \mathbf{a}_{0,i}^{(2)} &= \frac{-6(\mathbf{a}_{0,i} - \mathbf{a}_{1,i}) - \Delta t_i(4\dot{\mathbf{a}}_{0,i} + 2\dot{\mathbf{a}}_{1,i})}{\Delta t_i^2} \\ \text{and} & \\ \mathbf{a}_{0,i}^{(3)} &= \frac{12(\mathbf{a}_{0,i} - \mathbf{a}_{1,i}) + 6\Delta t_i(\dot{\mathbf{a}}_{0,i} + \dot{\mathbf{a}}_{1,i})}{\Delta t_i^3}. \end{aligned} \tag{3.7}$$

Finally, the new position and velocity of the integrated particles at time $(t_j + t_{last})$ are calculated using equation 3.8:

$$\begin{aligned} \mathbf{x}_i(t_i + \Delta t_i) &= \mathbf{x}_{p,i} + \frac{\Delta t_i^4}{24}\mathbf{a}_{0,i}^{(2)} + \frac{\Delta t_i^5}{120}\mathbf{a}_{0,i}^{(3)}. \\ \text{and} & \\ \mathbf{v}_i(t_i + \Delta t_i) &= \mathbf{v}_{p,i} + \frac{\Delta t_i^3}{6}\mathbf{a}_{0,i}^{(2)} + \frac{\Delta t_i^4}{24}\mathbf{a}_{0,i}^{(3)}. \end{aligned} \tag{3.8}$$

After the correction, the new t_j is calculated. This is the next time the particle has to be integrated. This is done for each particle for which the new acceleration and jerk has been calculated. When all integrated particles have been corrected the simulation program controls if the end time is reached or that more steps have to be taken.

3.1.2 Barnes-Hut treecode

The previous described method is a so called direct integration scheme that offers high accuracy, but has a complexity of $O(N^2)$. Therefore researchers have come up with an other integration method with a lower complexity. The treecode algorithm developed by Barnes & Hut in 1986 [2] has a complexity of $O(N \log N)$, but has lower accuracy than direct integration.

The treecode algorithm groups particles that are positioned close to each other together in a branch of the tree. This tree is created by iterating over all particles and then assign them to the various branches. The underlying structure of the tree is a quadtree and each quad has a limited amount of particles that it can hold. The amount of particles that a quad can fit is controlled by the user and can be used to increase / lower the accuracy of the simulation. When all particles are processed and assigned to one of the quads the algorithm will iterate over the quads. For each quad the centre of mass (*com*) and the global mass (*gb*) of the particles in the quad is calculated.

To integrate the whole system one step ahead in time, the algorithm iterates over each particle i. For each particle is decided if the force has to be calculated between i and the separate particles that are contained in the quad or between i and the quad as a whole using *com* and *gb*. This decision is made by using

equation 3.9 to calculate ratio r and then see if r is smaller than a user specified value (usually called Theta).

$$r = \frac{\text{size of quad}}{\text{distance from particle to centre of mass of quad}}. \tag{3.9}$$

The algorithm reduces the total number of force calculations as particles far away are seen as one big particle which speeds up the algorithm because fewer forces have to be calculated.

In summary, the Barnes-Hut treecode algorithm takes the following steps:

1. Build the quadtree.
2. For each quad in the quadtree, compute the centre of mass and total mass of all the particles it contains.
3. For each particle, traverse the tree to compute the force on it.

3.2 GRAPE

A breakthrough in direct-summation N-body simulations came in the late 1990s with the development of the GRAPE series of special-purpose computers [19] The GRAPE achieves spectacular speedups by implementing the entire force calculation in hardware and placing many force pipelines on a single chip. The latest special purpose computer for gravitational N-body simulations, GRAPE-6, performs at a peak speed of about 64 TFLOP/s [16]. The GRAPE opened the way for the simulation of large star clusters. In simulation software such as `starlab`, for example, the GRAPE is used as a coprocessor for the force calculations [33].

The GRAPE-6 chip is a hardware implementation of the force, jerk and potential formulas. The chip contains multiple pipelines that all calculate partial forces that are combined later on. The GRAPE-6 chip has 48 pipelines that calculate the forces in parallel. For each particle that is sent to the pipeline, the force, jerk and potential are calculated and for each particle is determined which of the other particles in the system is nearest and which particles are located within a specified radius. A GRAPE-6 board contains 32 chips that work in parallel. The partial results are combined using FPGA (Field Programmable Gate Array) chips.

The GRAPE-6Af is a commercial version of the GRAPE-6 module and contains 4 GRAPE-6 chips on a PCI board [9]. The GRAPE-6Af was designed to offer an alternative for the large and expensive GRAPE-6 boards that contain 32 chips. A single GRAPE-6Af chip can hold up to 131072 particles in memory and has a performance of about 123 GFLOP/s. We will compare our results with those obtained using a GRAPE-6Af board.

3.3 GPUs

The use of Graphics Processing Units (GPU) as an alternative to the GRAPE for use as a coprocessor in N-body calculations was first presented by Nyland et al. [27] and later improved by Mark Harris [13]. Their implementation only performs force calculations using a simplified shared time-step algorithm. A Cg implementation that performs force, jerk and potential calculations on a GPU through a block time-step algorithm was published by Portegies Zwart et al. [32]. They show that for large N the GPU offers an attractive alternative for the GRAPE-6Af because of its wide availability, low price and high reliability. Hamada et al. use CUDA to implement the force calculations, achieving an even higher performance [12], but their way of integrating the equations of motion by using shared time-steps is unsuitable for simulating dense stellar systems.

3.4 Implementation

3.4.1 Introduction

The implementation described here consists out of two parts. One part does the actual calculations on the GPU and the other part is the host code that runs on the CPU. The host code controls the GPU and is responsible for allocating memory, sending data to the GPU and starting executions. Besides the interaction with the GPU the host code also does the integration and the prediction and correction steps.

The next sections cover the details of the GPU application and explain how to get the maximum performance for the N-body problem. Section 3.4.5 covers the host code and explains what is done to ensure that the application on the GPU runs as fast as possible and how this all is combined into a library that can be used with existing software such as `starlab` [33].

3.4.2 Decomposition over CPU and GPU

In the test environment, the calculation of force, potential and jerk is performed on the GPU. The predictor and corrector steps are performed on the CPU. Our algorithm uses a block time-step scheme that only integrates subsets (blocks) of particles that need to be updated [21].

The decomposition of this scheme over a CPU and GPU was done for two reasons. First; the prediction and correction steps are more sensitive to round-off errors and are therefore best performed using the CPU's 64-bit floating point representation. Second; production quality software such as `starlab` [33] uses a similar decomposition, but then in combination with the GRAPE coprocessor.

We opted for a similar decomposition as used for the GRAPE to allow astronomical production software to link in our GPU implementation as a library.

Our implementation requires that particle data is communicated between the CPU and the GPU at each block time-step. This is accomplished through a number of memory copies where the CPU sends particle position, velocity and mass to the GPU. The results computed by the GPU (acceleration, jerk and potential) are retrieved by the CPU. For the GPU library the prediction is performed on the CPU after which all particles are copied to the GPU. The GRAPE only has to send the updated particles and performs the prediction on the GRAPE hardware itself. This results in an overall lower performance for the GPU when compared with the GRAPE, because the overhead of the memory copies increases much more for the GPU than for the GRAPE.

The input and output variables exchanged with the GPU program are the following:

- Input: mass (N), position ($3N$) and velocity ($3N$),
- Output: acceleration ($3N$), jerk ($3N$) and potential (N).

All values are represented by single precision (32-bit) floating point values, which is the most precise representation offered by current generation GPUs. This adds up to 14 floats or 56 bytes per particle which results in a total capacity of approximately 14 million particles for the 768MB on-board memory available on the GeForce 8800GTX. This is a substantial increase in capacity compared to the GRAPE-6Af's maximum capacity of 131072 particles. This is also an improvement over the 9 million particles that could be stored with the Cg implementation [32]. A restriction imposed by Cg that does not allow memory areas to be readable and writable at the same time forced this implementation to use a double-buffering scheme. This restriction does not exist in the CUDA implementation described here.

3.4.3 Base Implementation

The fundamental structure of the implementation aims at exploiting the available computing resources as much as possible. The challenge in mapping our N-body problem on a GPU through CUDA is to annihilate wait states due to slow memory accesses while keeping the threads executing on the GPU occupied.

Global memory access is slow while shared memory access is fast but has a limited capacity. We therefore pre-cache particles into shared memory up to its maximum capacity before the calculation of forces. Therefore the input data is split in smaller parts that are each pre-cached and processed in consecutive bursts.

The integration of one block time-step is initialized by assigning a thread to

each of the particles in a block. Each thread then goes through the following steps:

1. Each thread in the bundle caches one particle from global memory into shared memory. Each thread in a bundle reads one particle so that the total number of read particles is equal to the number of threads contained in a bundle.
2. The force, potential and jerk for the thread are calculated using the particles that are cached in shared memory. The thread then sums the partial results into temporary variables which are stored in a register.
3. Steps 1. and 2. are repeated until all particles have been read.
4. When all parts are processed, the self interaction of the potential value is removed, the results are saved in global memory and the thread exits.

Note that the total number of calculations performed by the GPU with this scheme is N^2. Although it is possible to determine the force using $\frac{1}{2}N(N-1)$ calculations, this would require internal communication and synchronization. This added communication is costly in a GPU and would result in lower performance even though less work is done.

The number of bundles that is started depends on the number of particles in the current time-step block. Each bundle in our implementation contains 128 threads (see section 3.4.6 for an explanation). Therefore the force, jerk and potential of 128 particles is calculated in parallel. In comparison; the GRAPE-6Af does the same but for 48 particles. The number of bundles that are started is equal to the number of particles in the time-step block divided by 128. This reduces the number of global memory accesses by a factor 128. Our implementation uses the thread scheduler to swap in threads that have already loaded their data while threads that are waiting on memory loads are swapped out. Once all threads have loaded the particle data from global memory into the shared memory space of the bundle, all threads in the same bundle can operate on that data. Through this strategy, the latency incurred by global memory accesses is hidden, which speeds up the algorithm considerably.

A stream diagram of the above described implementation can be found in Fig. 3.1. In this figure the stream diagram is mapped on top of a memory lay-out which indicates where the data resides that is required for the current action.

3.4.4 Optimizing GPU utilization

The implementation described in § 3.4.3 has the disadvantage that it does not utilize all processors in the GPU when the number of particles in a block time-step is smaller than 4096. This number is derived as follows: To make full use of all 16 multiprocessors in the GPU it is necessary to start at least 16

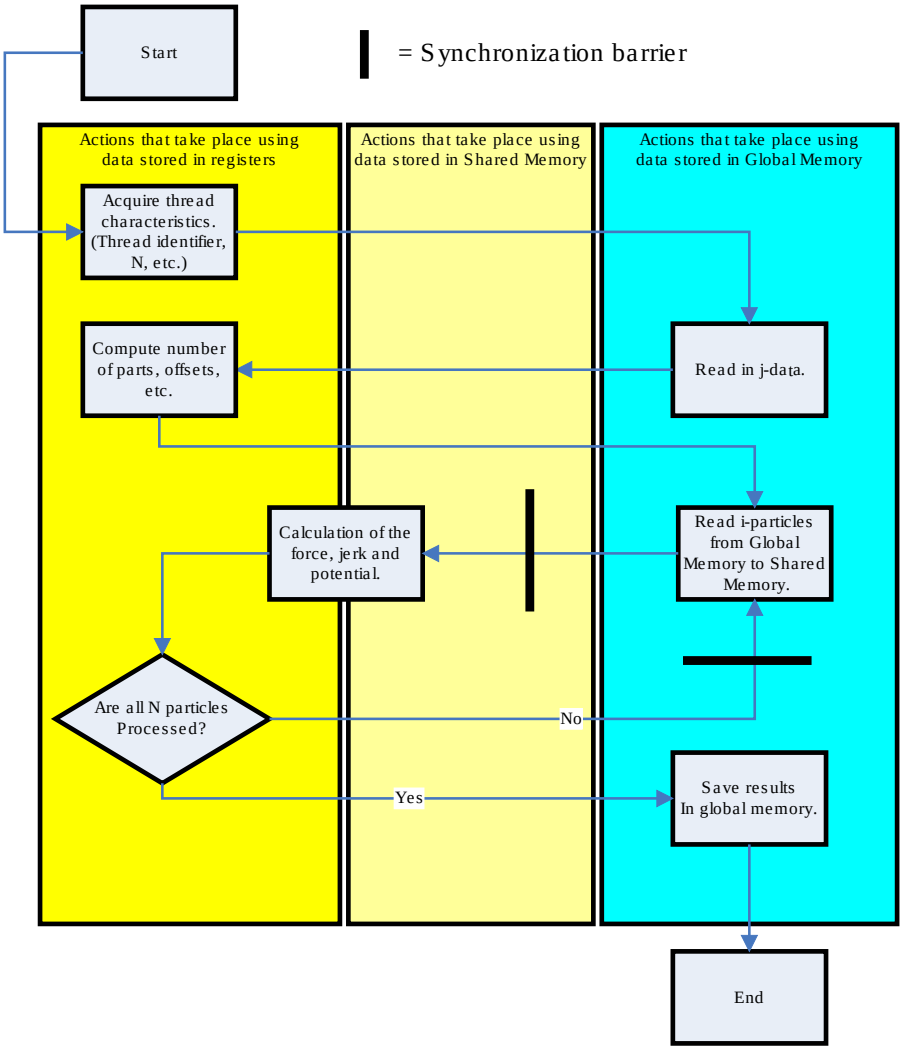

Figure 3.1: The stream diagram of the algorithm mapped on the memory architecture of the G80. The diagram shows the steps each thread takes. First loading of thread parameters and obtaining of the thread specific j-data. Then the actual force calculations start by loading data from global memory to shared memory. When all threads in the block have read the i-data and the synchronization is complete, the force is calculated using data in shared memory and in the registers. Then check if there is more data to process or that the process is finished and can write back the force, jerk and potential.

bundles. Moreover, threads in a bundle that are waiting for data from global memory will be swapped out in favour of bundles for which the data is ready and can be processed, which brings the total number of bundles to 32. With our implementation, where we start 128 threads for each bundle, we must have at least $32 \times 128 = 4096$ particles in the block time-step to fully utilize all 16 multiprocessors.

To fully utilize the GPU for any number of particles in the block time-step, we have altered the implementation in such a way that it splits the calculations in several parts and then combines the partial results on the host. This is done when there are less than 4096 particles in the block time-step.

The implementation divides the total number of particles in several parts that are processed sequentially. Each part contains 128 particles; equal to the number of threads per bundle. One by one the threads in each bundle load a particle i from global memory and then process the particles j that have been loaded in shared memory. When we have less than 4096 particles in the block time-step, the parts that have to be processed are evenly distributed, as much as possible, over multiple bundles. Each thread calculates a partial force between its particle i and the particles j in the part(s) that have been loaded from global memory. The partial results are then saved in global memory. This strategy assures that all multiprocessors in the GPU are fully utilized. As threads in different bundles cannot communicate it is not possible to aggregate partial results from finished bundles. Therefore the partial results are saved in global memory and subsequently combined on the host CPU. The host CPU loads the partial results from the GPU and then adds the partial results together.

The method described here can easily be adapted for use with other N-body problems. The only thing that needs to be changed are the equations in the inner loop. The data loading and processing algorithm stays the same.

3.4.5 Host Code

The host code is executed on the host computer and controls all interaction with the GPU. Interaction consists out of copying of data between the host memory and the GPU, initializing memory and starting of the threads on the GPU. This section explains what the host code does and what is done to ensure that the threads can execute without having to care about memory sizes and particle numbers.

Device Initialization The first thing that has to be done is to check if there is a CUDA capable GPU installed. If the driver can not find a suitable GPU then the program will exit. If the driver finds at least one suitable GPU then the first one is selected and used to perform the calculations. For future extensions it would be possible to parallelize the code in such a way that it can run on

multiple GPUs at the same time, if there is more than one GPU installed in the system.

If a device is available we calculate the number of particles that we will use in the system. The program receives the amount of particles that is used by the data set of the user. This number is increased to make it a multiple of the number of threads per bundle (*tpb*). This to guarantee that the particles can be split in equal sized parts; the size of each part is equal to the number of threads per bundle. Since all parts have a fixed size there is no need to check on the GPU if a memory read goes out of bound of the particle arrays. Since there is no need for a check, the GPU does not have to execute an expensive "if" statement that checks if memory reads fall out of range. The only cost added is the calculation of the force for some extra particles which have a mass equal to zero. Particles that have zero mass exercise zero force and have no influence on the final result.

After the total number of particles is calculated, the allocation of memory for the various arrays on the host and on the device is started. When the memory is initialized, the input arrays for the position and velocity are bound to textures. Although the textures do not directly increase performance, it makes it easier to extend the program to use 2D textures which might improve the performance of the memory reads. Besides the possible performance improvement, textures are also used to work around a bug in the compiler that has to do with reading data from global memory into dynamically allocated shared memory.

Data Transfers When the allocation of memory is finished, data can be copied from the host memory to the GPU memory. For each particle in the system, the mass, position and velocity is sent to the GPU. This data is represented as three arrays: one 1D array and two 2D arrays of which the second dimension has size 3 (x,y,z). If needed, a number of "dummy particles" is added to the arrays to make the number of particles a multiple of the number of threads (see above). When the data is ready to be sent to the GPU, three memory copies are executed which send the arrays from host memory to device memory.

The last data that is sent are configuration parameters. They are stored in the constant memory space of the GPU. The configuration parameters are required for each thread and are stored in constant memory because this memory space is cached, which results in faster access than when the values would have been stored in global memory. Configuration values are; the total number of particles N, the value that indicates the number of extra bundles that are started for the improved implementation and the softening value ϵ.

For each time step the complete data set is sent to the device instead of only the updated particles as is done in the GRAPE. This is done because of two reasons: one it is more efficient to execute one large memory transfer instead of multiple small transfers (see § 4.2.1 for details). The second and most important reason is that the prediction is performed on the host system. The force,

jerk and potential are calculated using these predicted positions and velocities. Therefore the particles on the GPU need to be updated with the newly calculated prediction values and therefore all data has to be sent to the device. When the prediction would take place on the device, only the position and velocity of updated particles has to be sent.

To retrieve the results, data is copied back from the device to the host. The results are saved in two arrays which both are copied from the device to the host. The first contains the calculated forces and the second contains the jerk and potential. The number of particles that is copied from the device to the host is equal to the number of particles in the block time-step (n). The results are then copied into result arrays of the calling program.

When the improved implementation is used, a bit more work has to be done to retrieve the results. First of all: more data has to be copied from the device since the results are split over multiple partial results. When all data is copied back from the device then the partial results need to be combined on the host. When the combining of the results is finished, the final result will be returned to the calling program. The cost for the extra work on the host is much smaller than the gain for using the improved implementation to calculate the forces.

Starting the computations When the particle data is loaded into the device the force computations can be started. The number of particles for which the force is calculated is n and since a thread is started for each particle in the time step-block, n threads need to be started. These threads are grouped in bundles and each bundle contains tpb threads. The number of bundles that is started is I, with: $I * tpb \geq n$.

To calculate I, equation 3.10 is used. Notice that I is increased by one if there is a partial division. This results in an increase of the total number of force calculations, but these results are ignored and the overall performance is increased since all necessary threads can be started in one GPU call. When no extra threads would be used, then the execution would be split in two separate GPU calls or extra "if" statements would have to be placed in the GPU code.

$$I = \begin{cases} (n/tpb) + 1 & \textbf{if } (\ n \mod tpb \neq 0) \\ n/tpb & \text{otherwise} \end{cases} \tag{3.10}$$

For the improved implementation the work is split over more than the previous I bundles to optimize GPU utilization. The number of bundles that is started needs to be high enough to fully utilize the GPU. For the 8800GTX the minimum number of bundles that is used is 32. The 8800GTX has 16 multiprocessors so 32 bundles results in 2 bundles per multiprocessor which enables the GPU to swap bundles to hide memory latency.

If $I < 32$ the work that the I bundles do is split over more bundles. The new number of bundles is a multiple of I so that each bundle does the same amount

of work.

The next three situations show what happens for various I:

If $I = 1$ then the number of bundles that is started is 32; this means that each bundle does 1/32 of the work it would do in the base implementation.
If $I = 31$ then the number of bundles that is started is 62; this means that each bundle does half of the work it would do in the base implementation.
If $I = 32$ then no extra bundles are started.

After the new number I is calculated, an extra configuration parameter is sent to the GPU. The parameter indicates how many extra bundles are started; this number is needed by the threads to be able to distribute the different parts over the different bundles.

Not all data sets can be split over 32 bundles. For example $N = 256$ can only be split over 2 bundles, with each bundle having to process 128 particles. The minimum size of the data set to fully utilize the GPU for each n is $N = 4096$[1]. Note that this is for $n \leq 128$.For larger n the size of the data set can be smaller to fully utilize the GPU since more bundles can be started. For example if $n = 256$ then the GPU can be fully utilized for data sets with $N \geq 2048$. [2]

3.4.6 Number of threads per bundle

Each bundle that is executed at the same moment on the GPU contains the same number of threads. We take $tpb = 128$, this is for the following reasons:

1. Threads on the G80 are started in groups of 32. Therefore to get full groups of threads, the number of threads preferably has to be a multiple of 32.

2. The amount of shared memory needed per bundle is directly related to the number of threads. If we start too many threads we can only have a few bundles concurrently active on a multiprocessor.

3. The occupancy (this is an indication of how much of the memory latency can be hidden) has a maximum with 128, as can be seen in figure 3.2.

4. Measurements show that 128 threads gave better performance than the other values in the figure. Partially because 2 of the peaks are no multiples of 32.

[1] Using 128 threads per bundle, to get a part for each thread there have to be at least 128 threads × 32 parts (bundles) = 4096 particles.

[2] The work can be split over 32 bundles, in the following way: The number of parts is 2048 / 128 = 16. The number of bundles that will be started with the base implementation is 256 / 128 = 2. Now the work of those 2 bundles is spread over 16 bundles, which results in 2 * 16 = 32 bundles in total.

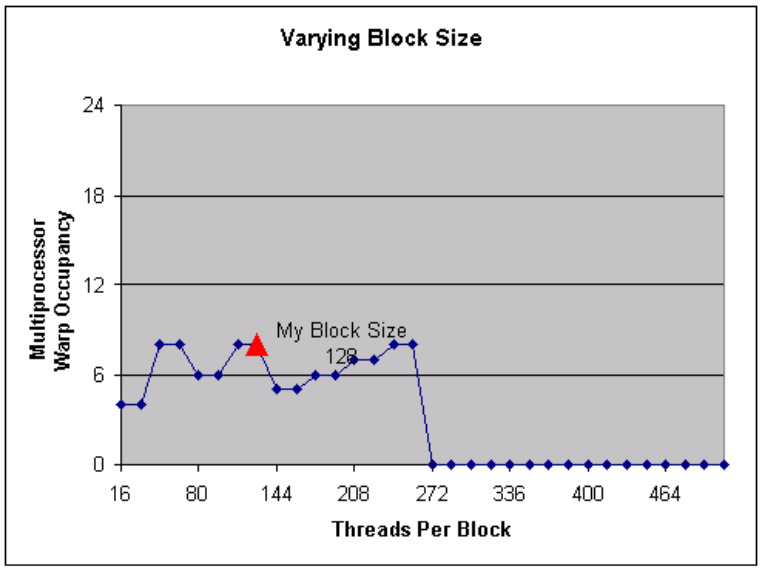

Figure 3.2: Results of the occupancy calculator, the triangle shows the peak at 128 threads.

3.5 Mimicking the GRAPE6 library

We have designed a library around our GPU based N-body code that mimics the standard GRAPE6 library. This allows existing applications that are linked to the GRAPE6 library to be used with `kirin` with minimal changes. Additional requirements are that the CUDA run-time libraries are installed on the system and that a graphics card capable of running CUDA applications is installed in the system. Appendix B shows a complete list of functions that have a GPU equivalent. The library is referred to as `kirin` library from now on.

3.5.1 Kernel changes

In addition to force, jerk and potential, the GRAPE hardware also calculates the nearest neighbour of every particle that is being updated, and the GRAPE has the ability to perform calculations without softening. The softening parameter ϵ, introduced by Aarseth in 1963 [1], prevents very small integration steps when particles reside very close to each other. The GPU code has to be adjusted to calculate the nearest neighbour and to handle simulations without softening.

Nearest Neighbour Nearest neighbours are determined by comparing the distance between each particle and all other particles in the data set. This is done as part of the force calculation; a comparison is added with each force calculation to maintain the particle with the minimum distance. When the force calculation is complete, the index to the nearest neighbour is saved in global memory, together with the force, jerk and potential results.

Handling softening The distance r_{ij} between two particles i and j can be zero either when $i = j$ or when the distance between two particles cannot be represented within the limited precision of a single precision floating point number. This results in a division by zero in the force equation (equation 3.3). The softening is added to the distance and has the effect that the distance between two particles can never be zero. For zero softening the resulting division by zero is circumvented by an additional check in the inner loop of the GPU program.

Adding each of these two comparisons results in lower performance: one extra comparison results in a performance drop of roughly 10%. This is mainly caused by the underlying SIMD architecture that enforces that when two threads take different branches, one has to wait until the branching thread reaches the same point in the program as the other.

This are the changes that have been made to the kernel compared to the test code, this goes for the normal implementation and the improved implementation. The performance will go down by roughly 20%.

3.5.2 Host part of the `kirin` library

Introduction

To make the library compatible with existing software there are a number of functions that need to be implemented. These functions are used by the official GRAPE library. The complete list of functions is quite extensive, with many functions having the same behaviour but with small changes to the parameter lists. Therefore only the most important functions are described here.

The GPU only has single precision which limits the precision of the calculations. Therefore the prediction of the particles takes place on the CPU instead of the GPU. In a previous version of the library the prediction took place on the GPU, but this resulted in very poor precision. To be able to do the prediction on the CPU, the library needs to keep a local copy of all particles that have been stored using `g6_set_j_particle`. The copy is kept up to date and will be used as source for the predictions.

The following GRAPE functions have a GPU equivalent:

1. g6_open, opens the connection with the GRAPE, for the GPU library it will initialize local buffers but not open the connection with the GPU yet.
2. g6_close, closes the connection with the GRAPE, for the GPU library it has as effect that it closes the connection with the GPU and releases all allocated memory. Since this function marks the end of the interaction with the devices, it will also free the memory that is allocated by the library.
3. g6_npipes, returns the number of pipelines that are on the chip (for the GRAPE this is 48). The GPU does not have a fixed number of pipelines, therefore the number can be configured using a configuration file. Tests show that for some applications the code is slowed down if the number of pipes is set too high.
4. g6_set_j_particle, sets a particle in the memory of the GRAPE, on the GPU the particle will be saved in a local buffer and send to the GPU after the prediction step. All these buffers are double precision arrays to keep an as high as possible accuracy. Besides the position, velocity and the mass of the particles we also store half of the acceleration, 1/6 of the first time derivative of the acceleration, 1/18 of the second time derivative and the time steps (last time the particle was updated and the step the particle takes). These values are stored and used for the prediction. The calling program specifies the address (1 to N) where the particle needs to be stored and its associated index number. Storing all these details of the particles results in relatively high memory usage for the library, but it is the only way to ensure that that the prediction of the particles takes place using double precision. Also it is the only way to send all particle

data to the GPU in one memory copy instead of many independent copies of single particles. Small copies results in bad performance since memory copies are "blocking" and have a relative high initialization time.

5. g6_set_ti, in the GRAPE library this sets the next time step to be used by the predictor on the GRAPE. In the GPU library it starts the predictor on the host system and will send the predicted particles to the GPU.

6. g6calc_firsthalf, on the GPU and GRAPE this has the same effect; the i-particles are received from the calling program and sent to the device. Then the force calculation for the particles specified in the function call will be started.

7. g6calc_lasthalf, on the GPU and GRAPE this has the same effect; the results of the previous g6_firsthalf call will be retrieved.

There are some other functions implemented as dummy functions that are just there for compatibility, since they do not have a meaning on the GPU. Examples are scaling of the precision and resetting of the hardware. The complete list can be found in Appendix B. This set of functions is enough to recompile programs that have been built with the basic GRAPE-6 library. There is however a more extensive library. If programs are built for this library then some support functions may have to be added. An example is the treecode implementation of Makino [17]. This only sends the particles position and mass to the GRAPE using a special mass and position only function. Extending the library by adding this function that only calls `g6_set_j_particle`, with the velocity set to zero, is enough to support the code. This method can be used for most other functions as well since the basic functions are implemented; it is only the special support functions that are missing.

3.5.3 *N*-body integration using the treecode

We have applied our `kirin` library to run with the treecode [2] as implemented by Makino [17]. This implementation has been designed to run on a GRAPE. Therefore we have linked the source code with our library to let the algorithm run on the GPU.

We adapted two different implementations of the library, the first is identical to the one described in § 3.5.2, the second one is optimized for the treecode. The Barnes-Hut treecode algorithm performs time integration using acceleration only, therefore the jerk and nearest neighbour's calculations can be left out. This results in a performance gain of a factor two (see Fig. 4.6). The direct integration method requires, besides the acceleration, also the derivative of the acceleration (jerk). Besides jerk, the `kira` integrator also requires the nearest neighbour of each particle that is integrated. Since the jerk and the nearest neighbour are not needed for the integration using the treecode, we can disable the code that calculates the jerk and the nearest neighbour to obtain extra performance.

3.5.4 Source code

The code for this project can be found online. The code includes all source code for the library (host and GPU code), Makefiles for Linux and Windows systems and documentation on how to build the library.

The code can be found on the following location:

`http://student.science.uva.nl/~jbedorf/kirin.zip`

Chapter 4

Results

4.1 Introduction

This chapter presents two types of results § 4.2 shows the results of various tests on the architecture which show the implications of the various design decisions and the differences between the base implementation and the improved one. Section 4.3 shows the results of N-Body integration using direct integration and treecode.

The results of N-Body integration will show the execution duration of the simulations for the various architectures (GPU, GRAPE6-Af and CPU) and the obtained energy error. Results for the test environment, GRAPE mimicking library and treecode are shown for various configurations. The library is used in combination with the `kira` integrator (which is part of `starlab`) and with the treecode implementation of Makino [17].

For the test environment a comparison is shown between the base implementation and the improved implementation to see the performance gain for small data sets. The simulations made with `kira` and treecode only make use of the improved implementation so no comparison is shown.

The data sets used are Plummer spheres [31] with all particles having an equal mass. The number of particles ranges with $N = 256$ up to $N = 1$ million, with size doubling each set (256, 512, 1024, ...). The Plummer spheres are scaled to virial equilibrium[1] before the integration is started. The direct integration simulations are run over 0.25 N-body time-step[2] and the treecode simulations are run over 1 N-body time-step.

The performance measurements in § 4.2 will show timings for memory copies and for calculation times only. The results that show the calculation time give

[1] see also `http://en.wikipedia.org/wiki/Virial_theorem`
[2] see also `http://en.wikipedia.org/wiki/Natural_units#N-body_units`

an indication for the performance of the GPU and show the difference between the base and improved implementation.

4.2 Architecture

The time required for interacting with the GPU can be split in two parts. The time required for the actual computations and the time required for communication with the GPU. The next two sections show the performance for the parts.

4.2.1 Memory Transfers

Timing measurements have been performed on sending particles to the GPU to back up the statements of section 3.4.5, where the implementation is discussed. It is mentioned there that sending particles in separate memory transfers is expensive. Therefore the time required for sending particles to the GPU is measured to determine the initialization time for memory transfers.

The time required for sending particles from and to the device is measured. This is done by sending a range of particles (starting at $N = 1$ up to $N = 2048$) to the device and measuring the time required to complete the transfer. Assumed is that one particles takes up 28 bytes (see § 3.4.2 for details) and is sent using one memory transfer. In the actual implementation multiple memory transfers are issued to place each characteristic (position, velocity and mass) of the particle in the correct memory location. Measuring multiple memory transfers however will result in the same characteristic.

The tests are performed using the bandwidth test program that comes with the CUDA SDK (SDK version 0.81). This program is used instead of the `kirin` application since the NVIDIA implementation removes cache dependencies. The plotted results are the average of executing the test program 4 times. The test program measures the duration of the memory copies for a range of data sizes. Each memory copy is executed 10 times to reduce the influence of random system behaviour. In total each test sample is executed 40 times. The times reported in the figure are the durations of executing the memory copy 10 times[3]. In Fig.4.1 the duration for the various memory transfers is presented, the two additional horizontal lines show the time required for sending two or three particles with each send / memory transfer containing only one particle. This gives a good indication of the relative high initialization time of the memory transfers.

[3] In ∼1% of the measurements the duration took 10 times longer than average, these outliers have been removed from the data. They are probably caused by random system behaviour, like interrupts, read or write faults, etc.

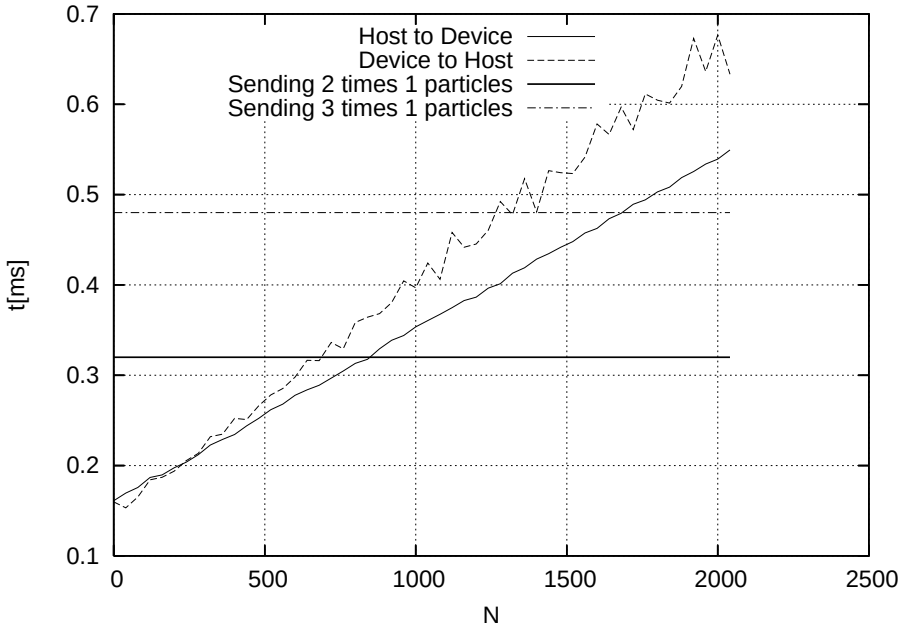

Figure 4.1: Duration measurement for sending data from and to the GPU. The x-axis represents the number of particles that is sent and the y-axis shows the duration in milliseconds (ms). The duration of sending data to the device is plotted as the dashed line. The duration of sending data from the device to the host PC is plotted as the solid thin line. The duration of sending two particles using two separate memory transfers is represented by the thick solid line. The duration of sending three particles using three separate memory transfers is represented by the dashed-dot line.

4.2.2 Computational Performance

The performance results of § 4.3 show the duration of a complete integration over a specified period of time. This includes the time required for prediction, integration, correction and the various memory transfers. In the next set of tables and figures only the time required for the actual force, jerk and potential calculations are shown. Not taken into account is the time required for sending the data to the device or for retrieving the results from the device. A comparison is presented between the base and the improved implementation to clarify the "need" for the improved implementation for small time-step blocks.

In Tab.4.1 the results for calculating the force, jerk and potential for all particles in the data set are shown. Calculating this for the whole data set with size N results in N^2 force, jerk and potential calculations. In the code used for these measurements there is no nearest neighbour calculation and softening can not

be set to zero.

As mentioned above this is only the time required for the calculation, not to load the particle data on the device or the time needed to retrieve the results. The table 4.1 shows the required time and the number of GFLOP/s this equals. The performance (P) in floating point operations per second (FLOP/s) is calculated using:

$$P = kN^2/t. \tag{4.1}$$

For the calculation of force, potential and jerk $k = 60$, as used by Makino et al. in [24, 23].

The table shows the results for the base implementation as well as the improved implementation. Note that the improved implementation does not create extra bundles when the data sets contain more than 8192 particles. The performance we measure here can be read as the performance that a shared time-step algorithm would reach. A shared time-step algorithm executes N^2 force calculations each time-step which is exactly what we measure.

Table 4.1: Performance measurements when calculating force, potential and jerk. The first column indicates the number of particles. The second and third column show the execution time and performance in GFLOP/s calculated using equation 4.1 with $k = 60$. The fourth and fifth column show the same as the second and third, but now for the improved implementation.

N	`kirin base` $[s]$	Performance GFLOP/s	`kirin improved` $[s]$	Performance GFLOP/s
256	0.00021	18.72	0.000132	29.79
512	0.00051	30.84	0.000134	117.7
1024	0.00095	66.23	0.000336	187.25
2048	0.00162	154.77	0.001149	219.02
4096	0.00453	221.97	0.004541	221.73
8192	0.01768	227.76	0.017731	227.10
16384	0.07004	229.97	0.070028	229.99
32768	0.27956	230.45	0.279635	230.39
65536	1.12463	229.15	1.1245	229.17
131072	4.46981	230.61	4.4686	230.67

In table 4.1 the performance difference between the two implementations using a shared time-step method was shown. The focus of this project was direct integration using a block time-step integration scheme, therefore also the performance difference between the two implementation using block time-steps is shown. As before only the time required for the actual calculations is taken into account. Three different data sets are used, with $N = 1024$, $N = 8192$ and $N = 131072$ particles.

The force calculation is executed for a range of block time-step sizes. With $n = 128$ [4] and doubling n up to $n = 8192$.

There is no need to do the simulations for more than 8192 particles, since both implementations have the same performance for $n \geq 8192$. The results can be found in Fig. 4.2.

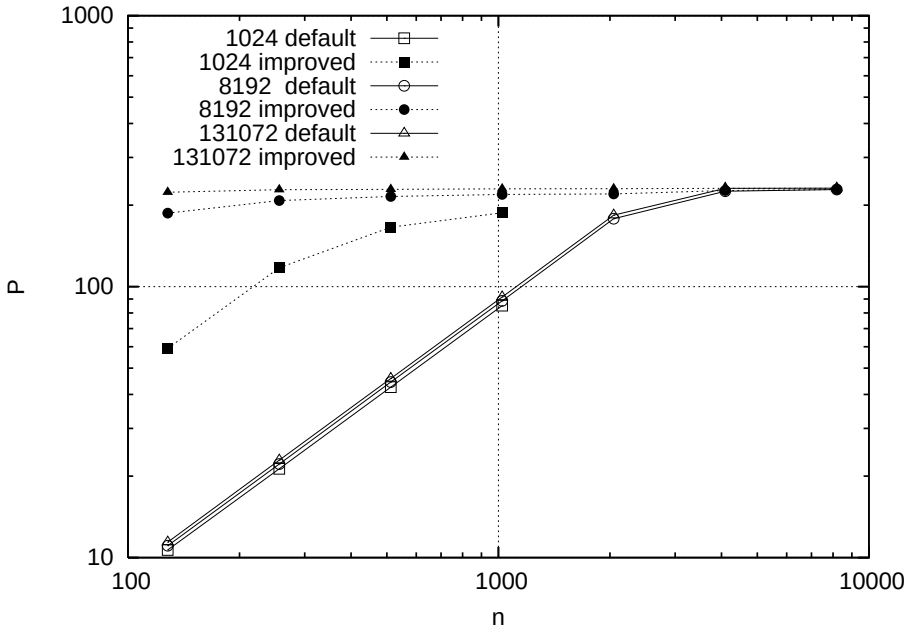

Figure 4.2: Performance measurements when calculating the force, jerk and potential using block time-steps for three different data sets. The x-axis represents the size of the block n. The y-axis represents the performance P in GFLOP/s. A comparison is made between the base implementation (solid lines) and the improved implementation (dotted lines). The $N = 1024$ data set is represented by the square symbols. The $N = 8192$ data set is represented by the circles and the $N = 131072$ data set is represented by triangles.

The last performance results show the minimum performance the GPU achieves for a block time-step algorithm. The performance for various data sets (starting with $N = 256$ up to $N = 1048576$) is measured using one block time-step that contains $n = 128$ particles. Again the performance of the base and improved implementation are plotted in the same figure to give an idea of the difference in performance that is achieved when using the full GPU instead of only 1/16th. The results can be found in Fig. 4.3.

[4] Starting at $n = 1$ will results in the same duration since the force of at least 128 particles is calculated.

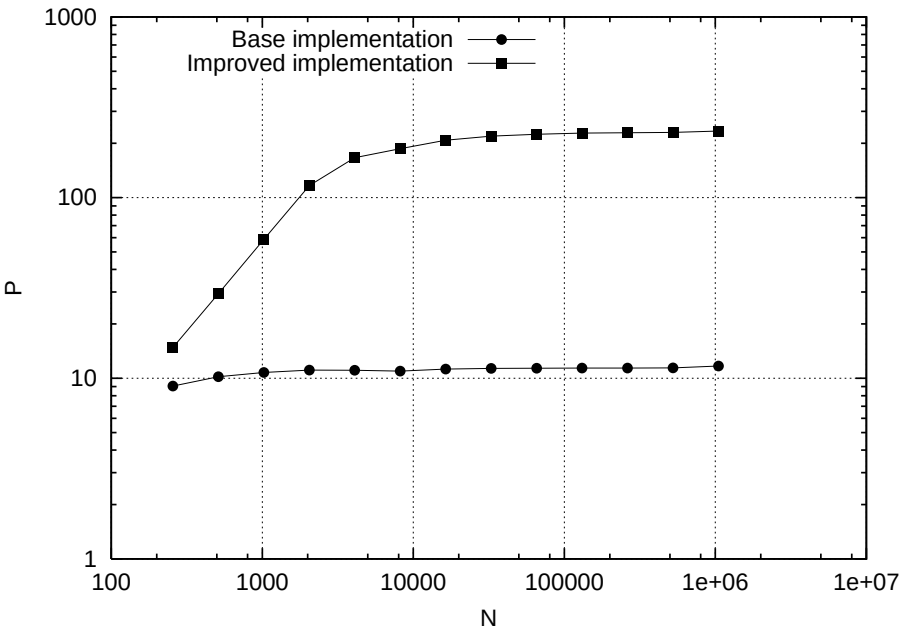

Figure 4.3: Performance measurements when calculating the force, jerk and potential using block time-steps for a fixed block size with $n = 128$. This shows the minimum performance that is reached for the base and improved implementations. The base implementation is represented by the solid line with circles. The improved implementation is represented by the solid line with squares.

4.3 Performance / Timing of N-Body codes

4.3.1 Test Code

The results presented here show the durations of the simulation using the test code. The test code contains a block time-step integrator in a custom framework. The GPU code calculates the acceleration, jerk and potential. No checks are done to see if the ϵ-value is zero (setting it to zero will result in wrong results) and there is no nearest neighbour calculation. The prediction, correction and time step approximation are all performed by the framework. The results show the duration to integrate the system from T=0.25 to T=0.5, with an ϵ-value of $\frac{1}{256}$ with an initial accuracy parameter of 0.01 and an accuracy parameter of 0.03.

The GRAPE results are limited to data sets up to 65536. This is caused by the fact that the PCI card can not hold more particles. Usually a card can hold up to 131072 or even 262144 particles, but one of the memory chips on the card is broken and therefore can only hold up to 65536 particles. The host results are

limited to 65536 for practical reasons, running even larger data sets would have taken a non-practical amount of time. The results can be found in Tab. 4.2 and Fig. 4.4.

Table 4.2: Performance of `kirin` compared to other implementations. The first column (N) gives the number of equal mass particles of a Plummer sphere. Columns 2 to 6 show the performance of the different implementations. The GRAPE-6Af column shows the result on GRAPE hardware, `kirin base`, `kirin improved` and the Cg implementation on the NVIDIA GeForce 8800GTX. The last column shows the performance of an implementation that ran completely on the host, an Intel Xeon at 3.4 GHz clock. The simulations were run over 0.5 N-body time step (timing measurements were done from $t = 0.25$ to $t = 0.5$). Some measurements are performed for limited N for practical reasons. The results on the GRAPE are limited to up to 65536 because of a defective memory chip.

N	GRAPE-6Af $[s]$	`kirin base` $[s]$	`kirin improved` $[s]$	Cg $[s]$	Xeon $[s]$
256	0.07098	0.136	0.107	2.708	0.1325
512	0.1410	0.382	0.217	8.777	0.5941
1024	0.3327	0.689	0.297	17.46	2.584
2048	0.7652	1.698	0.588	45.27	10.59
4096	1.991	4.013	1.646	128.3	50.40
8192	5.552	11.23	4.631	342.7	224.7
16384	16.32	31.11	14.28	924.4	994.0
32768	51.68	72.81	41.16	1907	4328
65536	178.2	183.8	129.8	3973	19290
131072	-	520.9	417.6	8844	-
262144	-	1771	1522	22330	-
524288	-	6040	5627	63960	-
1048576	-	20789	19975	-	-

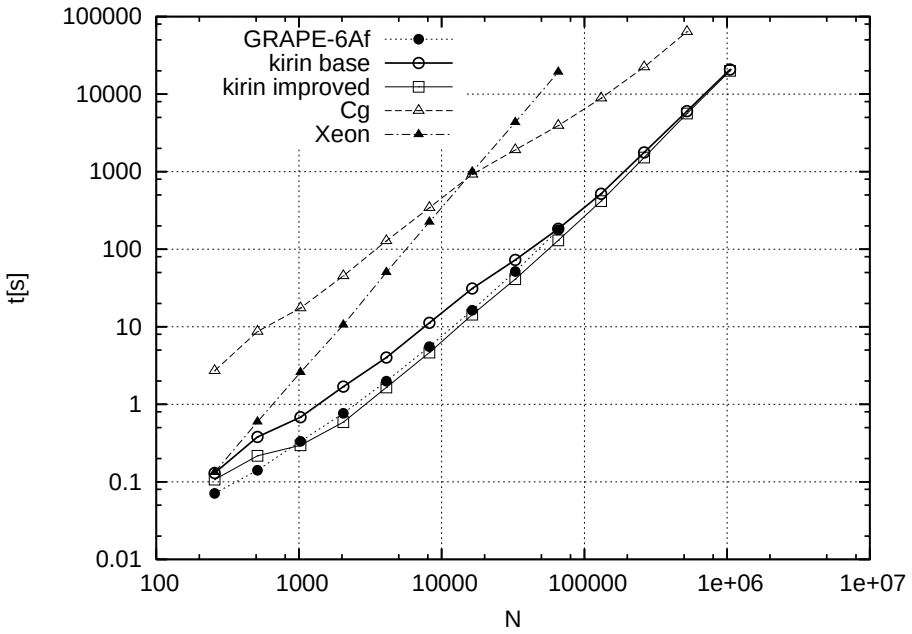

Figure 4.4: Performance comparison of the N-body implementations from Table 4.2. `kirin base` is represented by the solid line (open circles). `Kirin improved` is represented by the solid line (open squares) The GRAPE is represented as the dashed line (bullets). The Cg implementation is represented as the dashed line (open triangles). The dashed-dotted line (closed triangles) represents the results on the host computer.

4.3.2 Starlab using GPU library

The results presented in this section show the execution duration of the simulation using `starlab` with the `kira` integrator. Two versions of `starlab` are used, one that makes use of the GRAPE-6Af hardware and the other uses the GPU by making use of the `kirin` library. Besides the necessary computations for the integration the library can handle an ϵ-value of zero and calculates the nearest neighbour. The prediction is performed by the library, but the correction and the calculation of the next time-step is performed by `kira`.

The same settings are used for both versions with the exception of the number of pipes. The number of pipes is fixed for the GRAPE hardware, namely 48. For the GPU this can be an arbitrary number, but tests show that 16384 give the best performance. It is more efficient for the GPU to have a larger number of particles to work on, but for `starlab` it would slow down the `starlab` code too much if larger values then 16384 were used. For 16384 we almost obtain the maximum amount of performance for our GPU implementation for all n (see Fig. 4.3). A larger number of pipes will not give much more GPU performance and will only slow down the `starlab` code.

The settings used for `starlab` where an accuracy parameter of 0.3 (default value is 0.1) and softening either set to $\epsilon = 1/256$ or to $\epsilon = 0$. The simulations are run over one fourth N-body time step. As with the test code the results of the GRAPE are limited to data sets of sizes up to 65536.

In Tab. 4.3 and Fig. 4.5 the performance of the GRAPE and the GPU for N-body simulations using `starlab` is shown.

Table 4.3: Performance measurements comparing execution time of the standard GRAPE6 library with our GPU library. The tests are performed by using the `starlab` software package. Columns 2 and 3 show the GRAPE and GPU results with $\epsilon = 1/256$. Columns 4 and 5 show the results of the same simulation with $\epsilon = 0$.

	$\epsilon = 1/256$		$\epsilon = 0$	
N	GRAPE-6Af $[s]$	`kirin` $[s]$	GRAPE-6Af $[s]$	`kirin` $[s]$
256	0.06	0.12	0.06	0.11
512	0.11	0.22	0.13	0.19
1024	0.27	0.29	0.27	0.39
2048	0.65	0.54	0.67	0.74
4096	1.65	1.51	1.79	3.75
8192	4.33	4.35	4.7	8.57
16384	12.02	11.17	13.18	20.2
32768	35.69	32.5	41.4	57.1
65536	116.1	101.1	146	202
131072	-	355	-	735
262144	-	1313	-	2668
524288	-	4913	-	11190
1048576	-	18681	-	46372

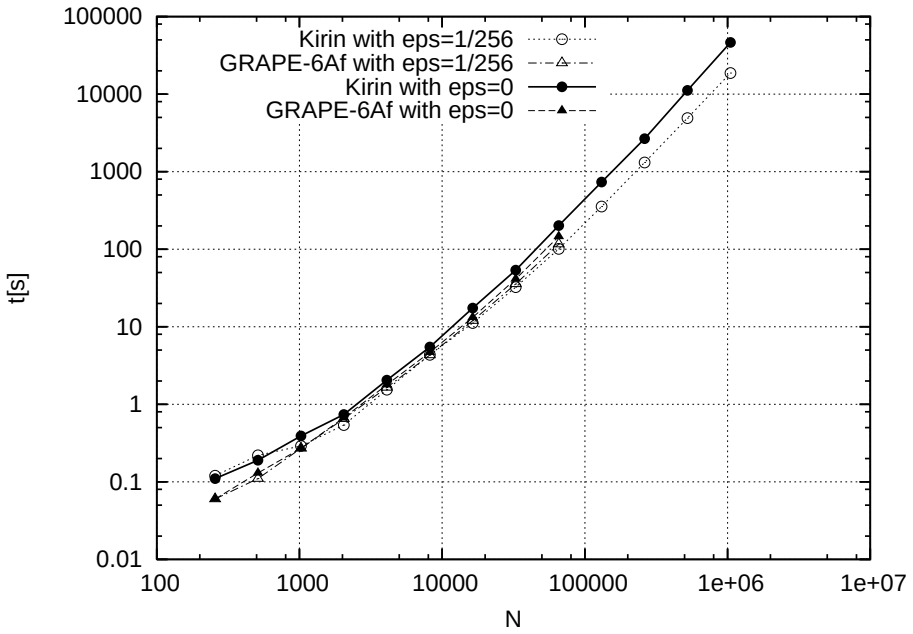

Figure 4.5: Performance comparison of the N-body implementations from Table 4.3, using the `kira` integrator in `starlab`. The `kirin` library with $\epsilon = 1/256$ is represented by the dotted line (open circles). The same library with $\epsilon = 0$ is represented by the solid line (bullets). The standard GRAPE6 library with $\epsilon = 1/256$ is represented by the dash-dotted line (open triangles). The same library with $\epsilon = 0$ is represented by the dashed line (closed triangles).

4.3.3 Treecode

For the treecode measurements the implementation developed by Makino is used [17].

The code is linked with the `kirin` library to run it on the GPU. No code changes are required, besides the size of some of the arrays that have been raised from 96 to 32768. The default value is related to the number of pipelines of the GRAPE-6 (there are GRAPE-6 boards that contain 96 pipelines). The new size is set to 32768, because after various test runs with multiple data sets it turned out that 16384 or 32768 pipelines gave the best results. Therefore the upper number is taken so that both settings can be used depending on the simulation data.

The treecode simulations are performed over 1 N-body time unit, using the GRAPE-6Af, GPU and the CPU of the host system. For the GPU simulations two different versions of the library are used, the default one that is used with `starlab` and one that is optimized for treecode. The optimized version does not calculate the jerk and nearest neighbour since these are not required for the algorithm.

The results of the treecode simulations are presented in Tab.4.4 and Fig. 4.6. All runs are performed using the same default settings with the exception that the number of pipes and the ncrit value for Barnes' vectorization (the "ncrit" value controls the average number of particles in a group). The values are set such that the best performance for each specific data set is achieved. The same settings per data set are used for the GRAPE, the GPU (normal and optimized) and the CPU.

Table 4.4: Performance measurements comparing the execution time of the treecode using the standard GRAPE6 hardware, the GPU and the CPU. For the GRAPE and GPU we choose an "ncrit" value of either 8192, 16384 or 32768; whichever was fastest . Other than this, all simulations are run over 1 N-body time unit with default settings.

N	GRAPE-6Af $[s]$	`kirin (normal)` $[s]$	`kirin (optimized)` $[s]$	CPU $[s]$
256	0.85	0.40	0.39	0.34
512	1.25	0.47	0.46	0.78
1024	0.71	0.59	0.57	1.61
2048	2.69	0.85	0.79	3.58
4096	5.07	1.58	1.28	8.27
8192	10.7	3.77	2.65	19.9
16384	23.9	10.2	5.57	45.6
32768	51.4	16.9	11.7	104
65536	109	42.3	25.4	249
131072	266	117	59.9	564
262144	682	379	169	1230
524288	1033	563	394	2752
1048576	2004	1247	733	5985

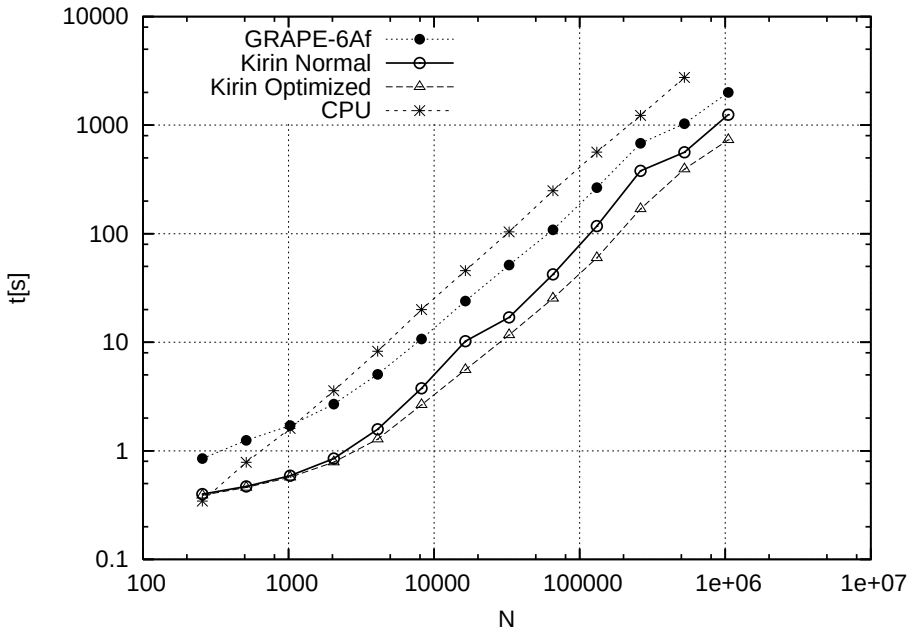

Figure 4.6: Performance comparison of the execution time of the treecode from Table 4.4 over 1 N-body time unit. The GRAPE hardware is represented with the dotted line (bullets), the normal version of the GRAPE mimicking library is represented as the solid line (open circles). The optimized version of the library is represented as the dashed line (open triangles). The CPU is represented as the dashed line (stars).

4.4 Relative Energy error

In an isolated system the total energy must remain constant. Therefore the relative energy error is used to measure the accuracy of the implementation in relation to the GRAPE. The relative error $\triangle E/E$ is calculated using equation 4.2.

$$\triangle E/E = \frac{E_{start} - E_{end}}{E_{start}}. \tag{4.2}$$

Here E_{start} is the energy before the simulation over time, E_{end} is the energy of the system after the simulation.

The error is related to the precision of the integration and the softening ϵ value. The following results show the relation between the relative error and the step size of the integrator. It will show that a smaller step size increases the accuracy, however a smaller step size results in longer execution times which will be shown in the results. The measurements have been done by making use of `starlab` in combination with the `kirin` library. The GRAPE-6Af and the GPU are compared in these simulations.

The first set of results shows the relation between the relative error size and the step size of the integrator.

The average energy errors for various simulations can be found in Tables 4.5 and 4.6. The average relative energy error is the average of the simulations for $N = 256$ to $N = 65536$. There have been 6 different simulations, we have varied the accuracy a of the integrator and have run simulations with softening set to $\epsilon = 1/256$ and without softening $\epsilon = 0$. The integration steps used are $a = 0.3$, $a = 0.2$ and $a = 0.1$ (which is the default setting of `starlab`).

Table 4.5: The relative energy error $\triangle E/E$ of the simulations performed with `kira`. The first column (N) gives the number of equal mass particles of a Plummer sphere. Columns 2 to 7 show $\triangle E/E$ for the GRAPE and the GPU using $\epsilon = 1/256$. The relative error was obtained by running the simulation over 0.25 N-body time unit using 3 different accuracy parameter $a = 0.3$, $a = 0.2$ and $a = 0.1$. The other parameters where the same as used in the measurements for Table 4.3.

N	G6Af $a = 0.3$ $[\times 10^{-7}]$	`kirin` $a = 0.3$ $[\times 10^{-7}]$	G6Af $a = 0.2$ $[\times 10^{-8}]$	`kirin` $a = 0.2$ $[\times 10^{-8}]$	G6Af $a = 0.1$ $[\times 10^{-9}]$	`kirin` $a = 0.1$ $[\times 10^{-9}]$
256	1.14	0.40	0.42	1.00	0.28	20.1
512	0.33	0.40	0.68	3.00	0.56	23.8
1024	0.25	0.78	0.39	3.58	0.97	29.1
2048	0.21	0.31	1.08	2.50	0.98	19.3
4096	8.71	8.92	1.59	2.75	0.58	9.33
8192	51.5	51.5	1.72	2.58	1.54	12.9
16384	3.75	3.46	3.80	4.25	2.04	6.92
32768	8.32	8.14	1.34	3.37	2.39	21.4
65536	37.0	37.3	2.24	2.12	3.72	4.95
Average	12.35	12.25	1.48	2.79	1.45	16.42

Table 4.6: The relative energy error $\triangle E/E$ of the simulations performed with `kira`. The tests performed for this table are basically the same as executed for Table 4.5, but now without softening. The first column (N) gives the number of equal mass particles of a Plummer sphere. Columns 2 to 7 show $\triangle E/E$ for the GRAPE and the GPU using $\epsilon = 0$. The relative error was obtained by running the simulation over 0.25 N-body time unit using 3 different accuracy parameter $a = 0.3$, $a = 0.2$ and $a = 0.1$. The other parameters where the same as used in the measurements for Table 4.3.

N	G6Af $a = 0.3$ [$\times 10^{-7}$]	`kirin` $a = 0.3$ [$\times 10^{-7}$]	G6Af $a = 0.2$ [$\times 10^{-8}$]	`kirin` $a = 0.2$ [$\times 10^{-8}$]	G6Af $a = 0.1$ [$\times 10^{-9}$]	`kirin` $a = 0.1$ [$\times 10^{-9}$]
256	0.11	2.00	0.49	1.55	0.90	0.38
512	0.73	0.01	0.79	0.17	0.78	3.29
1024	0.53	0.91	0.65	1.25	0.90	6.35
2048	0.16	0.13	1.64	3.01	0.93	19.4
4096	10.9	11.6	1.45	1.96	0.47	9.26
8192	151	151	0.90	2.18	1.14	22.0
16384	86.1	86.2	4.18	4.83	1.58	12.4
32768	497	498	124	122	1.67	19.2
65536	1421	1413	72.9	74.1	1.86	0.70
Average	240	240	2.30	2.35	1.14	10.3

The previous tables show the relation between the relative energy error and the step size of the integrator, but the execution time is not shown. The execution time changes when the step size of the integrator is changed. A smaller step size of the integrator leads to longer execution times, especially for the GPU where the need for higher accuracy is achieved by taking even smaller time-steps than is done on the GRAPE.
To show the effect of the integration step size on the execution time and relative energy error, the results of the simulations for two different data sets have been plotted in one figure. The two selected data sets are one with $N = 1024$ particles and the other has $N = 16384$ particles. Again the simulations have been performed with softening set to $\epsilon = 1/256$ and with zero softening $\epsilon = 0$. The accuracy parameter a is taken between 0.3 and 0.025 with steps of 0.025. The results with softening set to $\epsilon = 1/256$ can be found in Fig. 4.7. The top half represents the time required for the simulation over 0.25-N-body time step for the various accuracy settings. The bottom half shows the relative energy error. The results with softening set to $\epsilon = 0$ can be found in Fig. 4.8 . The top half represents the time required for the simulation over 0.25-N-body time step for the various accuracy settings. The bottom half shows the relative energy error.

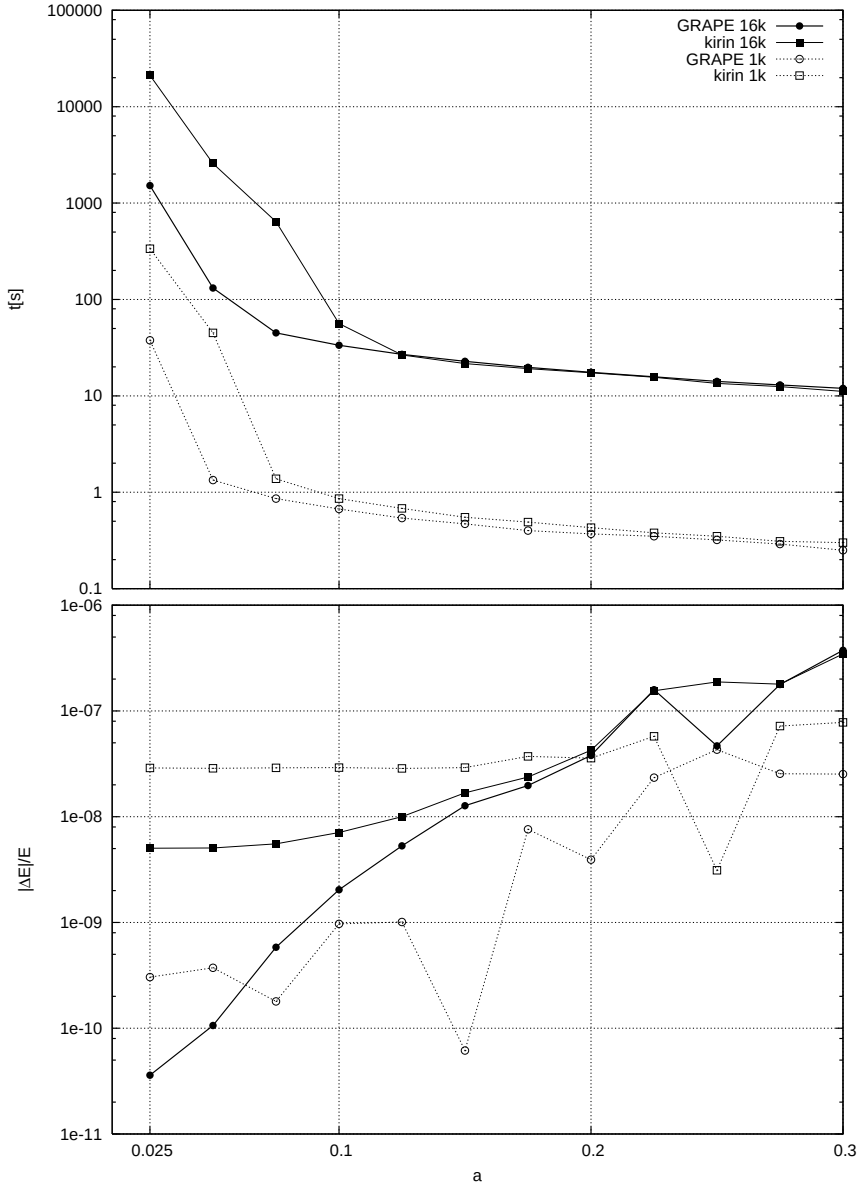

Figure 4.7: Performance and relative error comparison of the `kirin` library and the default GRAPE library using `kira` using softening ($\epsilon = 1/256$). For two data sets (1024 and 16384 particles) the execution time and relative energy error is plotted. The execution time is plotted in the upper half of the image, the energy error is plotted in the bottom half. The simulation is executed over 0.25 N-body time-step with $\epsilon = 1/256$. The 1024 data set is represented by the dashed line and the 16384 data set is represented by the solid line. The GRAPE library is represented by the circles, solid for the 16384 data set and open for the 1024 data set. The `kirin` library is represented by the squares, solid for the 16384 data set and open for the 1024 data set.

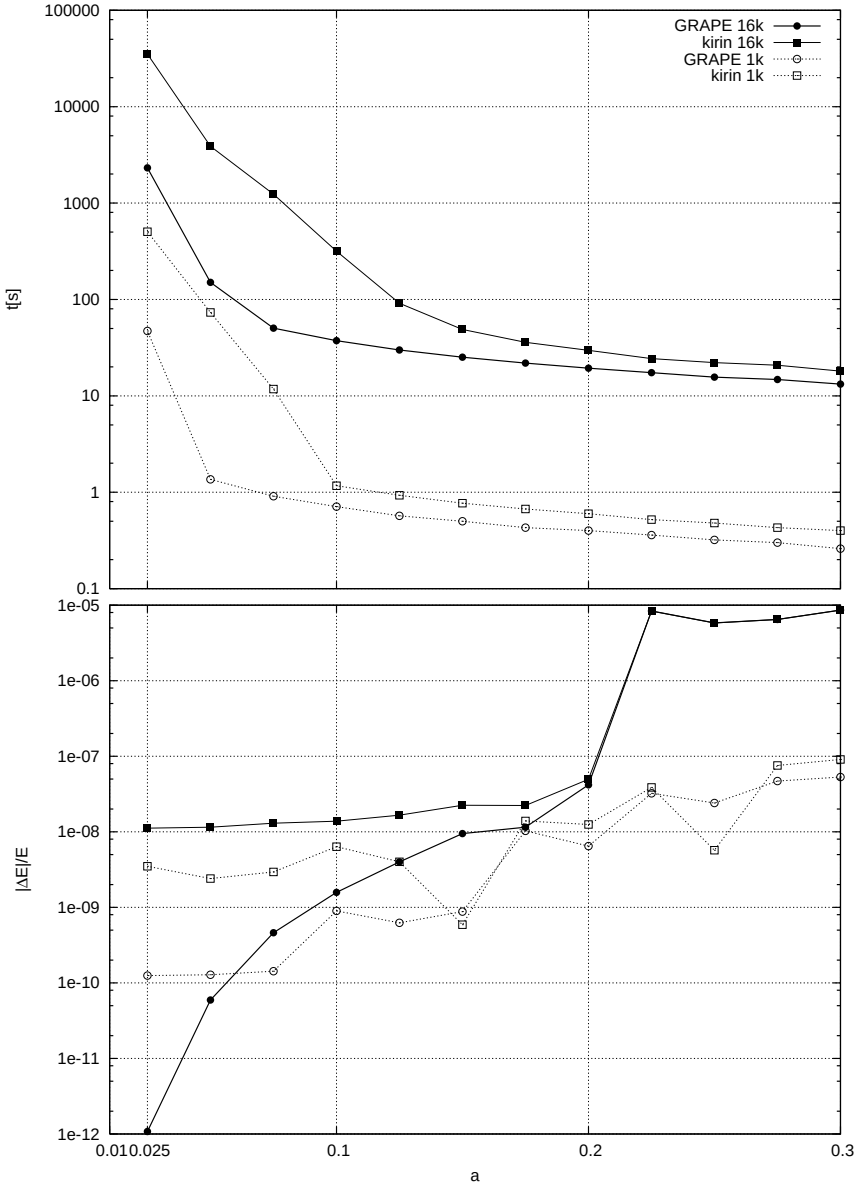

Figure 4.8: Performance and relative error comparison of the `kirin` library and the default GRAPE library using `kira` using zero softening ($\epsilon = 0$). For two data sets (1024 and 16384 particles) the execution time and relative energy error is plotted. The execution time is plotted in the upper half of the image, the energy error is plotted in the bottom half. The simulation is executed over 0.25 N-body time-step with $\epsilon = 0$. The 1024 data set is represented by the dashed line and the 16384 data set is represented by the solid line. The GRAPE library is represented by the circles, solid for the 16384 data set and open for the 1024 data set. The `kirin` library is represented by the squares, solid for the 16384 data set and open for the 1024 data set.

Chapter 5

Discussion

5.1 Architecture

Memory

The results of Fig.4.1 indicate that the initialization time for 10 memory transfers is ~ 0.16ms.

The consequence of the relatively high initialization time is that the duration of sending two particles using two separate memory transfers takes as long as sending ~ 850 particles in one memory transfer. With the current hardware all particles have to be sent to the GPU during each time-step (because prediction is on the CPU), so the time required for sending separate particles is not that important. However, when double precision GPUs become available, the prediction can be performed on the GPU. If the prediction is done on the GPU, the duration of separate memory transfers will be something that has to be taken into account. It is not required anymore to send all N (predicted) particles to the GPU, since only the location and velocity of the n updated particles is changed. Therefore only these updated particles have to be sent to the GPU (as is done with the GRAPE hardware). Now the tricky part is to determine what the best method would be to send the (updated) particles to the GPU.

When each particle is sent using a separate memory transfer then we pay the price of the relative high initialization time for each particle. When we send all N particles then the price is paid for a large memory transfer while a small one would be sufficient. The better solution would be to cache the updated particles on the host and then send them in one batch to the GPU. The particles are then stored in a temporary buffer and a small GPU program copies all particles in the correct memory locations. When double precision GPUs are available, further testing has to be performed to see if this results in the expected performance or that sending particles in separate memory copies is faster after all.

5.1.1 Computational Performance

The computational performance results as presented in Table 4.1 show the performance for calculating the force, jerk and potential. The performance that the kernel used in the test environment achieves is ∼230GFLOP/s. This kernel does no nearest neighbour approximation and can not handle zero softening. The difference between the base and the improved implementation is noticeable for the smaller data sets where the improved implementation reaches a higher performance since the computations are spread over more multiprocessors in the GPU. For data sets with $N \geq 4096$ the performance for the base and improved implementation is the same, since both implementations start enough bundles to keep the GPU busy and hide as much memory latency as possible.

The results of Table 4.1 show the performance in a shared time-step environment, but the main goal of this project was to make an implementation suitable for block time-steps. In the block time-step algorithm the force, jerk and potential is calculated for n particles each time-step. Usually n is a lot smaller than N, therefore the implementation has to offer good performance for each block size n. The results of the performance for various block sizes using three different data sets can be found in Fig. 4.2. The results show that the implementation (by making use of the improved kernel) manages to get a performance of ∼230GFLOP/s for each n when a data set is used with size $N = 131072$. For smaller data sets the performance depends on n, for very small data sets $N \leq 1024$ the peak performance will not be reached since there is not enough data to keep the GPU busy while hiding memory latency.

Finally, Fig. 4.3 shows the minimum performance the GPU achieves for different sized data sets. Again the base and improved implementation are compared for reference. The size of the blocks is fixed to $n = 128$ which is the minimum number of particles that is processed. The implementation reaches the peak speed for data sets that have size $N \geq 16384$. For smaller data sets there is not enough data to hide memory latency. The difference between the base and the improved implementation is clearly visible and is caused by the fact that the improved implementation starts more bundles to occupy all the GPU resources. The base implementation only starts one bundle which results in the use of only one multiprocessor of the 16 available and since there is only one bundle, the GPU can not switch between bundles to hide memory latencies.

5.2 Performance

5.2.1 Test Code

The previous chapter presented the results for the integration of a Plummer sphere using the block time-step algorithm. The results indicate that the GPU has the same or even better performance than the GRAPE hardware, for data

sets that contain more than 512 particles. For smaller data sets the algorithm can not make full use of all the GPU resources which results in lower performance. With the configuration that is used only one fourth of the GPU is used if the data set contains 512 particles. For a data set with 256 particles it is one eighth. When all of the GPUs resources are used (e.g. when the data set contains more than 512 particles) the GPU outperforms the GRAPE hardware. This is caused by the fact that the GPU has more processing power than the GRAPE.

The performance difference between the base and improved implementation becomes smaller when the size of the data sets increase, as is visible in Fig.4.4 and Tab.4.2. This is caused by the fact that for large data sets there are generally more particles in the block time-step, therefore the number of times the improved implementation is needed occurs less often. Therefore there is less room for improvement since the algorithm already makes optimal use of the GPU for most blocks. For smaller data sets like the one with 8192 particles the effect of the improved implementation is clearly visible since the speedup is more than a factor 2.

The difference in performance between the host and the GPU / GRAPE is clearly visible. The time needed on the host computer increases a lot faster than the time needed on the special hardware implementations. The GPU outperforms the host computer with a factor 100 for data sets with size $N \geq 65536$.

5.2.2 The GRAPE like library

The results presented in Tab.4.3 and Fig.4.5 show that the performance of the GPU is in line with the performance of the GRAPE hardware when softening is set to $\epsilon = 1/256$. When the simulation is run without softening ($\epsilon = 0$) the GRAPE will be faster than the GPU for all N.

Compared to the kernel used in the test environment, the library requires more particles to be faster than the GRAPE. This is caused by the extra computations that are required for nearest neighbour approximation and the fact that the library code can handle an arbitrary softening ϵ-value.

That the GRAPE is faster than the GPU when zero softening is used can be attributed to the following three causes:

1. When zero softening is used the branch inside the kernel code on the GPU results in two threads that will execute two different pieces of code when the distance between particles i and j is zero. Since executing two different pieces of code causes one of the threads to stall, this results in longer execution time since there will be times that the threads have to wait for each other.

2. Zero softening results in more occurrences of close encounters. The GPU calculates the distance using single precision, while the GRAPE does this in double precision. This results in that on the GPU the distance is more often calculated as being zero than on the GRAPE. Therefore there occur more calculation errors on the GPU which the integrator tries to solve by taking smaller time-steps. Smaller time-steps mean that the average size of n goes down. If n is smaller than 128, the GPU code performs more force calculations than are actually needed[1].

3. A smaller time-step results in more time-steps in total. For each time-step we have to predict all particles of the data set. The GRAPE does this in parallel on the device, while the GPU implementation does this on the host CPU. Since the GPU implementation does this on the host CPU, all these predicted particles have to be copied to the device (GPU). This results in more communication overhead for `kirin` compared to the GRAPE.

5.2.3 Treecode

The results of Tab.4.4 and Fig. 4.6 show that the GPU outperforms the GRAPE and the CPU for all data sets. This goes for both the normal and optimized library. That the GPU outperforms the GRAPE can be attributed to the fact that the treecode algorithm works different than the predictor-corrector algorithm. More particles are integrated at the same time which makes the GPU a lot more efficient since less memory copies are required. Memory copies are expensive for the GPU implementation since the data for all (predicted) particles is sent to the device, even if only one particle is updated. With treecode both the GPU and the GRAPE implementations have to send a large number of particles to the hardware. So the GRAPE has no notable advantage anymore with respect to memory transfer time.

5.3 Relative Error

5.3.1 Error size

The energy is used to measure the accuracy of the integrator and the difference between the GRAPE results and the GPU results. Tables 4.5 and 4.6 show that for an accuracy setting of $a > 0.1$ the GRAPE and the GPU have an error of the same order. With $a = 0.1$ the relative error of the GRAPE is one order smaller than that of the GPU. This is caused by the use of double precision within the GRAPE chip. This occurs in the simulations that have been run

[1] As explained in § 3.4.6, always 128 threads are started. If n is for example 28 that means that the forces for 128-28=100 particles are calculated while they are not needed, which is a waste of computational power, but a direct consequence of the GPU architecture.

with softening and the ones without. The results indicate that the relative error for simulations without softening is larger than the simulations that have been run with softening for $a > 0.1$. This is caused by the fact that the removal of softening results in more calculations where the distance between two particles is very small which leads to more round off errors. If the accuracy is set to $a = 0.1$ the integration steps become so small that round off errors occur less often and the relative error becomes smaller. As with simulations with softening, the relative error of the GRAPE is one order smaller than that of the GPU.

5.3.2 Error size vs. execution time

Figures 4.7 and 4.8 show the relation between execution time, relative energy error and the accuracy of the integrator. The images show the advantage that the GRAPE has over the GPU when a small accuracy value is used. The GRAPE with its 64bit calculations achieves an accuracy that is one to two orders of magnitude better than that of the GPU while keeping the execution time an order of magnitude shorter. When an accuracy parameter between $a = 0.1$ and $a = 0.3$ is used, the CPU and GRAPE achieve results that are of the same order, but when a smaller accuracy parameter is taken the GRAPE becomes superior in speed and precision.

5.4 Multi GPU / System

One GPU is not sufficient enough to simulate large star clusters over an extended period of time. The same goes for the GRAPE6-Af. For example, simulating a million body simulation over 10.000 time-steps takes over two years to complete. The solution is to use multiple GPUs as is done with the large GRAPE6 machines that consist out of multiple GRAPE boards. Modern motherboards have support for installing two graphic cards (in some cases even three, but the third makes use of a slower connection) on the same board. This instantly results in a doubling of the available computational power available. To get even more computational power multiple machines can be used in combination with software like MPI to connect the machines and execute the program in parallel.

The `kirin` library allows us to let existing software that has originally been written to function with the GRAPE6 hardware be run on the GPU. Existing software is used to test the performance of making use of two machines that both have one GPU installed. The software used is the Gcopy program of Derek Groen (based on the code of Alessia [11]) and the treecode program that we have used before (§ 4.3.3). Both programs can be executed in parallel by making use of MPI.

The treecode uses the GRAPE/GPU hardware right from the start and executes all calculations in parallel as much as possible. Therefore the whole execution time is taken when calculating the speedup. The Gcopy program however does

the calculation of the initial energy on the host. For large data sets this initialization time contributes a considerable amount of the total execution time. Therefore two speed up values have been calculated; one with initialization time and one where the initialization time is removed from the total execution time, this results in the time needed for the simulation over time only.

In Table 5.1 and 5.2 the results of the simulations by making use of a multi-GPU configuration are presented. Plots of the tables can be found in Fig. 5.1 and Fig. 5.2

The results show that when enough particles are used the scaling is almost linear (at least in the case of 2 machines). One has to remember that the codes are not optimized for GPUs and especially the performance of the code of Derek could improve for small data sets when one decides to only use parallel machines when n is large enough to make optimal use of the GPUs. For the treecode the results for smaller data sets are better than those of the Gcopy and comparable for the larger data sets, namely a speedup of $\sim$1.9. The fact that the treecode is faster for smaller data sets is caused by the way treecode works, namely creation of various clusters that can be integrated individually / independently. The linear speed-up of the treecode is also observed by Makino [17] for two and more machines using GRAPE hardware.

The machines used at SARA are not suitable for installing two graphics cards in the same system since it requires a powerful power supply and the ones installed by SARA are not powerful enough to keep the whole system stable. Therefore we used two different machines, both with one GeForce 8800GTX installed (since there are only two systems with a GeForce 8800GTX available we could not do simulations using more than two machines).

Recently researchers have built a multi-GPU system consisting out of 16 machines with 2 GeForce 8800GTX graphic cards installed per system [34]. The theoretical peak performance this system offers is 16.2 TFLOP/s and they achieve a performance of 7.1 TFLOP/s. This is around 223 GFLOP/s per GPU. The high performance that is achieved for each separate card shows that the multi-GPU systems are very scalable. A GPU cluster also has another advantage over the GRAPE clusters; they can be used for other things than astrophysical calculations as opposed to the GRAPE which is only suitable for one specific task.

Table 5.1: Performance measurements of N-body integration on two machines in parallel. The simulations are run over 0.25-N-body time-step, using the built-in settings of the code: softening $\epsilon = 1/256$, accuracy of 0.02 (This accuracy value can not be compared to the one used in the `starlab` simulations.). Column 1 gives the number of equal mass particles of a Plummer sphere. Column 2 shows the total simulation time using one machine, columns 3 shows the total simulation time on two machines. Column 4 shows the relative speedup of using two machines instead of one machine. Column 5 and 6 show the results of the integration time only. This is the total time minus the initialization time. Column 7 shows the relative speedup of using two machines instead of one machine.

	Total time			Integration time only		
N	1 GPU $[s]$	2 GPU $[s]$	Speed up	1 GPU $[s]$	2 GPU $[s]$	Speed up
8192	5.11	6.15	0.83	4.19	5.22	0.80
16384	16.9	17.3	0.97	13.0	13.4	0.97
32768	62.9	57.4	1.10	45.7	40.2	1.14
65536	206	169	1.22	134	97.8	1.37
131072	750	583	1.29	461	292	1.58
262144	2875	2145	1.34	1711	981	1.75
524288	11207	8178	1.37	6569	3535	1.86

Table 5.2: Performance measurements of N-body integration on two machines in parallel using treecode. The simulations are run over 1 N-body time-step. Column 1 gives the number of equal mass particles of a Plummer sphere. Column 2 shows the duration of the simulation when one machine is used. Column 3 shows the duration when two machines are used. The last column shows the speed-up of using two machines instead of one.

N	1 GPU $[s]$	2 GPU $[s]$	Speed up
256	0.48	0.53	0.92
512	0.55	0.62	0.88
1024	0.67	0.64	1.06
2048	0.89	0.83	1.07
4096	1.34	1.22	1.10
8192	2.69	1.99	1.35
16384	5.66	3.38	1.67
32768	12.3	8.24	1.49
65536	26.1	15.9	1.64
131072	60.1	33.8	1.78
262144	169	90	1.87
524288	373	195	1.91
1048576	709	380	1.87

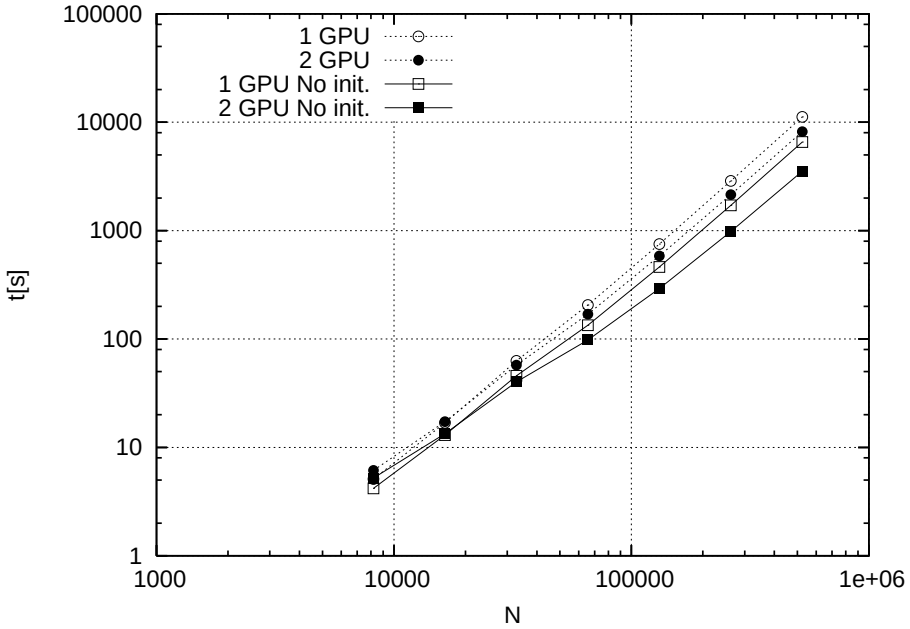

Figure 5.1: Performance measurements of N-body integration on two machines in parallel. Shown are the results when using one system and when using two systems with and without initialization time. The initialization takes place on the CPU of one system and there is no parallelism involved. The simulations are run over 0.25-N-body time-step, using the built-in settings of the code: softening $\epsilon = 1/256$, accuracy of 0.02. The duration on one GPU with initialization time is represented with the dotted line and open circles. The dotted line with closed circles shows the same simulation with initialization time included, but now on two GPUs. The durations where the initialization time is removed are represented by the solid line. The open squares show the duration on one GPU, the closed squares show the duration on two GPUs.

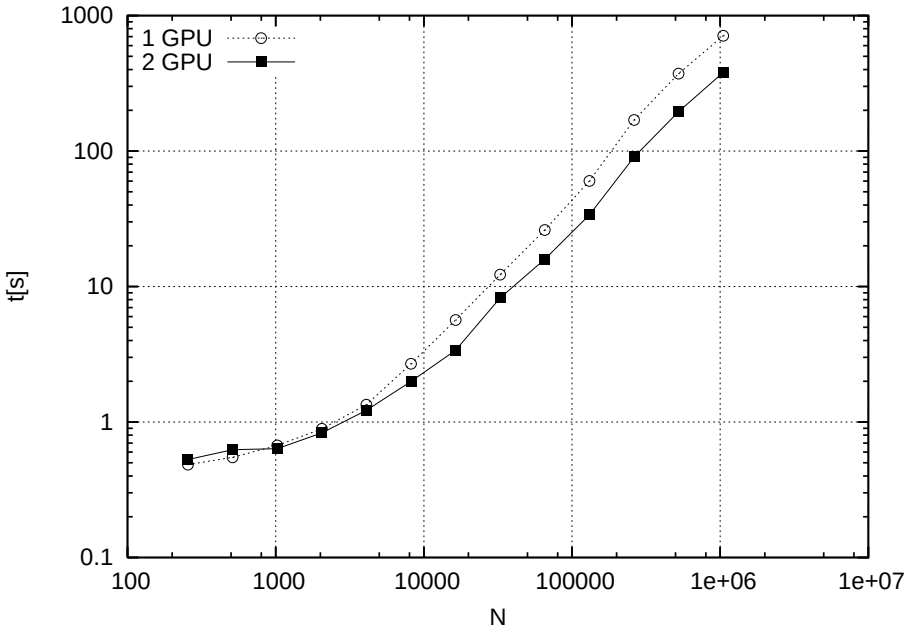

Figure 5.2: Performance measurements of N-body integration on two machines in parallel using the treecode algorithm. Shown are the results when using one system and when using two systems . The simulations are run over 1 N-body time-step. The dashed line with open circles shows the duration of the simulation when one machine is used. The solid line with filled squares shows the duration when two machines are used.

5.5 Recommendations for future work

CUDA The CUDA environment is still relative new and the NVIDIA engineers work hard to improve it. Improvement in register usage and more efficient memory transfers have lead to better performance for code that runs on the GPU since fewer resources are used. One of the most notable improvements in the CUDA libraries since the first versions is the introduction of asynchronous execution of the kernels. This means that code can be started on the CPU while there is a program running on the GPU. This way both the GPU and CPU can execute code at the same time which is truly parallel use of the GPU and CPU. A good use of this would be to do correction of particles while the forces of other particles are calculated. This requires reorganization of current code, but will result in performance increase especially when the time-step block contains a large number of particles. At the moment memory transfers are synchronized, but there are plans to make them asynchronous which results in being able to do memory transfers while the CPU is doing something else.

Configuration The performance of `kirin` depends on the amount of bundles and threads that are started. Since the optimal number of threads and bundles depends on the design (type/version) of the GPU, it is hard to provide an optimal value. The maximum number of threads that can be initialized cannot exceed the number of registers available to store the partial accelerations, jerks and potentials. The overall performance depends therefore on the number of registers available on the multiprocessors. Ideally CUDA should have a routine that returns the optimal number of threads and bundles. In CUDA 1.0 NVIDIA added functions to retrieve the various characteristics of the device like the number of registers per multiprocessor, total available memory, revision numbers of the hardware and some more details. Using these details it is possible to give a rough calculation of what would be the best configuration if the characteristics of each revision of the GPU would be saved in the program. This is not optimal, but at least it is better than the first versions of the CUDA SDK, that did not offer this functionality.

Double precision The most important area of improvement in GPUs is support for double precision. This requires a change in the hardware which is planned to be introduced late 2007. Double precision will be introduced on special GPUs in the new NVIDIA `Tesla` series [2]. This product line is specifically focused on general purpose computing. The cards are not even suitable for desktop use since they do not have a video output port. With the introduction of double precision the GPU will become a real alternative for specialized hardware like the GRAPE-6 machines. How the use of double precision will work out in terms of performance is something that is not known yet, but it is expected

[2] NVIDIA Tesla - GPU Computing Solutions for HPC - http://www.nvidia.com/tesla

that NVIDIA will introduce a product with roughly double the performance of the GPU used in this project (the G80).

Power consumption One of the major problems with today's GPUs is the power consumption and heat generation. The 8800GTX uses roughly ∼175W when running on peak speed and reaches temperatures above 80 degrees Celsius, which is a lot more than specialized hardware. It is hard to predict whether the next generation requires less power and runs cooler. Although the GPUs will probably be smaller [3] which would result in less heat generation, but this effect is negated by higher clock speed which results in more heat. The same goes for the power consumption, higher clock speeds and more transistors require more power. All in all the power consumption is something to keep in mind when assembling a computer system for High Performance Computing since it requires a power supply capable of providing enough electric power and have good enough cooling to keep the whole system stable.

[3] Produced using 65nm instead of 95nm used for the G80.

Chapter 6

Conclusion

We conclude, by looking at the results, that the latest generation of GPUs offer massive amounts of computational power that can be efficiently accessed using the CUDA environment. The presented GPU algorithm outperforms the CPU of the host by a factor of 100 and performs on comparable speed with the (much more expensive) GRAPE-6Af hardware, given that there are enough particles in the data set to reach the peak performance.

Graphics processing units offer an attractive alternative to specialized hardware, like GRAPE. While GPUs are programmable, however limited, they can be deployed for a wider range of problems, whereas GRAPE is single purpose. Also, the cost for purchase and maintenance of a GPU is much lower than for GRAPE. However, the single precision of current GPUs remains a problem, but should be solved when double precision GPUs are introduced. Note also that the GRAPE we used is the smallest 1-module PCI version, and obviously we cannot outperform a TFLOP/s GRAPE-6 board of the full 64 TFLOP/s GRAPE system with a single GPU. A GPU cluster on the other hand might be able to compete with the larger GRAPE clusters.

One also has to keep in mind that the performance of the GPU N-body implementation is a lot more sensitive for changes in accuracy of the integrator. When high accuracy is desired, the GRAPE hardware will be a better choice since it manages high performance at high accuracy. The GPU with its fixed 32bit floating point arithmetic on the other hand looses performance while barely improving accuracy. When one chooses to make use of treecode instead of direct integration the GPU becomes a real alternative for the GRAPE since the treecode with its lower accuracy requirements can be efficiently run on the GPU while achieving the same error size as on the GRAPE hardware.

The CUDA environment still has a lot of room for improvements. All in all we can conclude that the future for GPGPU (using CUDA) looks bright and will draw even more attention when the support for double precision will be added

to the hardware.

The code developed in this project is available for use by others. Recently, the code has been adopted and successfully modified by researchers at the Institute for Atomic and Molecular Physics (AMOLF, Amsterdam) for molecular dynamics simulations on a GPU [35].

Appendix A

A.1 Deviations from the IEEE-754 standard

Floating-Point Standard Compute devices follow the IEEE-754 standard for single-precision binary floating-point arithmetic. The following lists the deviations to the IEEE-754 standard as implemented on the G80. List taken from the CUDA documentation [26].

- Addition and multiplication are often combined into a single multiply-add instruction (FMAD);
- Division is implemented via the reciprocal in a non-standard-compliant way;
- Square root is implemented via the reciprocal square root in a non-standard-compliant way;
- For addition and multiplication, only round-to-nearest-even and round-towards-zero are supported via static rounding modes; directed rounding towards +/- infinity is not supported;
- There is no dynamically configurable rounding mode;
- Denormalized numbers are not supported; floating-point arithmetic and comparison instructions convert denormalized operands to zero prior to the floating-point operation;
- Underflowed results are flushed to zero; There is no mechanism for detecting that a floating-point exception has occurred and floating-point exceptions are always masked, but when an exception occurs the masked response is standard compliant; Signaling NaNs are not supported.
- The result of an operation involving one or more input NaNs is not one of the input NaNs, but a canonical NaN of bit pattern 0x7fffffff. Note that in accordance to the IEEE-754R standard, if one of the input parameters

to min() or max() is NaN, but not the other, the result is the non-NaN parameter.

The conversion of a floating-point value to an integer value in the case where the floating-point value falls outside the range of the integer format is left undefined by IEEE-754. For compute devices, the behaviour is to clamp to the end of the supported range. This is different behaviour then the x86 architecture.

Appendix B

B.1 Implemented GRAPE6 function

```
int         GPU_g6_close(int clusterid);
int         GPU_g6_open(int clusterid);
int         GPU_g6_npipes();
int         GPU_g6_open(int clusterid);
void        GPU_g6_set_ti(int cluster_id, double ti);
void        GPU_g6_initialize_jp_buffer(int clusterid, int size);
void        GPU_g6_flush_jp_buffer(int clusterid);
int         GPU_g6_reset(int cluster_id) ;
int         GPU_g6_reset_fofpga(int cluster_id);
void        GPU_g6_set_neighbour_list_sort_mode(int mode);
int         GPU_g6_set_j_particle(int clusterid,
                                  int address, int index, double tj, double dtj,
                                  double mass, double k18[3], double j6[3],
                                  double a2[3], double v[3], double x[3]);
void        GPU_g6calc_firsthalf(int cluster_id,
                                  int nj, int ni, int index[], double xi[][3],
                                  double vi[][3], double aold[][3], double j6old[][3],
                                  double phiold[], double eps2, double h2[]);
int         GPU_g6calc_lasthalf(int cluster_id,
                                  int nj, int ni, int index[], double xi[][3],
                                  double vi[][3], double eps2, double h2[],
                                  double acc[][3], double jerk[][3], double pot[]);
void        GPU_g6_set_tunit(int newtunit);
void        GPU_g6_set_xunit(int newxunit);

int         GPU_g6_set_j_particle_multisend_mxfast_(int * clusterid,
                                   int * nclusters, int *address,  int *index,
                                   double *mass,  double x[3]);

int         GPU_g6_flush_jp_buffer_and_multisend(int clusterid, int clusters);

int         GPU_g6_print_chip_status(int clusterid);
int         GPU_g6_reinitialize(int clusterid);


int         GPU_g6_open_(int *clusterid);
int         GPU_g6_close_(int *clusterid);
int         GPU_g6_npipes_();
void        GPU_g6_set_ti_ (int *cluster_id, double *ti);
int         GPU_g6_reset_ (int *cluster_id);
int         GPU_g6_reset_fofpga_ (int *cluster_id);

void        GPU_g6_set_tunit_(int *newtunit);
void        GPU_g6_set_xunit_(int *newxunit);
```

```
void        GPU_g6_set_overflow_flag_test_mode(int aflag, int jflag, int pflag);

void        GPU_g6_initialize_jp_buffer_(int *clusterid, int *size);
void        GPU_g6_flush_jp_buffer_(int *clusterid);

int         GPU_g6_set_j_particle_(int *clusterid,
                            int *address, int *index, double *tj, double *dtj,
                            double *mass, double k18[3], double j6[3],
                            double a2[3], double v[3], double x[3]);
void        GPU_g6calc_firsthalf_ (int *cluster_id,
                            int *nj, int *ni, int index[], double xi[][3],
                            double vi[][3], double aold[][3], double j6old[][3],
                            double phiold[], double *eps2, double h2[]);
int         GPU_g6calc_lasthalf_ (int *cluster_id,
                            int *nj, int *ni, int index[], double xi[][3],
                            double vi[][3], double *eps2, double h2[],
                            double acc[][3], double jerk[][3], double pot[]);


void        GPU2_g6calc_firsthalf_(int *cluster_id,
                            int *nj, int *ni, int index[], vec xi[], vec vi[],
                            vec aold[],vec j6old[], double phiold[],
                            double *eps2,  double h2[]);

int         GPU2_g6calc_lasthalf_(int *cluster_id,
                            int *nj, int *ni, int index[], vec xi[],
                            vec vi[], double *eps2, double h2[],
                            vec acc[], vec jerk[], double pot[]);

int         GPU2_g6calc_lasthalf2_(int *cluster_id,
                            int *nj, int *ni, int index[], vec xi[],
                            vec vi[], double *eps2, double h2[], vec acc[],
                            vec jerk[],double pot[], int nnbindex[]);

int         GPU_g6_get_neighbour_list_ (int *cluster_id,
                                        int *pipe, int *max_length,
                                        int *nblen, int nbl[]);
int         GPU_g6_read_neighbour_list_ (int *cluster_id);
```

Bibliography

[1] S. J. Aarseth. Dynamical evolution of clusters of galaxies, I. *Monthly Notices of Royal Astronomical Soc.*, 126:223–255, 1963.

[2] J. Barnes and P. Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. Nature, 324:446–449, December 1986.

[3] R. G. Belleman, J. Bédorf, and S. Portegies Zwart. High Performance Direct Gravitational N-body Simulations on Graphics Processing Units – II: An implementation in CUDA. *New Astronomy*, 13:103–112, February 2008. doi:10.1016/j.physletb.2003.10.071.

[4] David Blythe. The Direct3D 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[5] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[6] NVIDIA CUDA. CUDA Programming Guide Version 0.8, February 2007.

[7] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html, April 2007.

[8] R. Fernando. *GPU Gems (Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison Wesley, 2004.

[9] Toshiyuki Fukushige, Junichiro Makino, and Atsushi Kawai. GRAPE-6A: A single-card GRAPE-6 for parallel PC-GRAPE cluster system, 2005.

[10] P. Geldof. *Generic Computing on a Graphics Processing Unit*. MSc thesis, Universiteit van Amsterdam, May 2007.

[11] A. Gualandris. *Simulating self-gravitating systems on parallel computers*. PhD thesis, University of Amsterdam, September 2006. ISBN 978-90-6464-024-7.

[12] T. Hamada and T. Iitaka. The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units. *ArXiv Astrophysics e-prints*, March 2007.

[13] M. Harris. GPGPU: General-Purpose Computation on GPUs. In *SIGGRAPH 2005 GPGPU COURSE*, 2005.

[14] OpenGL Shading Language. http://www.opengl.org/documentation/glsl/, April 2007.

[15] J. Makino. A Modified Aarseth Code for GRAPE and Vector Processors. *Publ. Astr. Soc. Japan* , 43:859–876, December 1991.

[16] J. Makino. Direct Simulation of Dense Stellar Systems with GRAPE-6. In S. Deiters, B. Fuchs, A. Just, R. Spurzem, and R. Wielen, editors, *ASP Conf. Ser. 228: Dynamics of Star Clusters and the Milky Way*, page 87, 2001.

[17] J. Makino. A Fast Parallel Treecode with GRAPE. *Publ. Astr. Soc. Japan*, 56:521–531, June 2004.

[18] J. Makino and S. J. Aarseth. On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems. *Publ. Astr. Soc. Japan* , 44:141–151, April 1992.

[19] J. Makino and M. Taiji. *Scientific simulations with special-purpose computers : The GRAPE systems.* Scientific simulations with special-purpose computers : The GRAPE systems by Junichiro Makino & Makoto Taiji. Chichester ; Toronto : John Wiley & Sons, c1998., 1998.

[20] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[21] S. L. W. McMillan and S. J. Aarseth. An O(N log N) integration scheme for collisional stellar systems. *ApJ* , 414:200–212, September 1993.

[22] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[23] K. Nitadori, J. Makino, and G. Abe. High-Performance Small-Scale Simulation of Star Clusters Evolution on Cray XD1. *ArXiv Astrophysics e-prints*, June 2006.

[24] K. Nitadori, J. Makino, and P. Hut. Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a hermite scheme on x86_64 architecture. *New Astronomy*, 12:169–181, December 2006.

[25] NVIDIA. NVIDIA CUDA Programming Guide 0.8, February 2007.

[26] NVIDIA. NVIDIA CUDA Programming Guide 1.0, June 2007.

[27] L. Nyland, M. Harris, and J. Prins. N-body simulations on a GPU. In *ACM Workshop on General-Purpose Computing on Graphics Processors (Poster)*, pages C–37, 2004.

[28] J. Owens. Streaming Architectures and Technology Trends. In Matt Pharr, editor, *GPU Gems 2*, chapter 29, pages 457–470. Addison Wesley, March 2005.

[29] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[30] M. Pharr and R. Fernando. *GPU Gems 2 (Programming Techniques for High-Performance Graphics and General-Purpose Computation)*. Addison Wesley, 2005.

[31] H. C. Plummer. On the Problem of Distribution in Globular Star Clusters. *MNRAS*, 71:470, 1911.

[32] S. F. Portegies Zwart, R. G. Belleman, and P. M. Geldof. High-performance direct gravitational N-body simulations on graphics processing units I: An implementation in Cg. *New Astronomy*, 12:641–650, November 2007.

[33] S. F. Portegies Zwart, S. L. W. McMillan, P. Hut, and J. Makino. Star cluster ecology - IV. Dissection of an open star cluster: photometry. *MNRAS* , 321:199–226, February 2001.

[34] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, and T. Chiueh. Graphic-Card Cluster for Astrophysics (GraCCA) – Performance Tests. *ArXiv e-prints*, 707, July 2007.

[35] J. A. van Meel, A. Arnold, D. Frenkel, S. F. Portegies Zwart, and R. G. Belleman. Harvesting graphics power for MD simulations. *ArXiv e-prints*, 709, September 2007.