GPU-Accelerated Monte Carlo Simulations of Dense Stellar Systems

Bharath Pattabiraman*
bharath@u.northwestern.edu

Stefan Umbreit†
s-umbreit@northwestern.edu

Wei-keng Liao*
wkliao@eecs.northwestern.edu

Frederic Rasio†
rasio@northwestern.edu

Vassiliki Kalogera†
vicky@northwestern.edu

Gokhan Memik*
memik@eecs.northwestern.edu

Alok Choudhary*
choudhar@eecs.northwestern.edu

* Center for Interdisciplinary Exploration and Research in Astrophysics, Dept. of Electrical Engineering and Computer Science Northwestern University, Evanston, USA

† Center for Interdisciplinary Exploration and Research in Astrophysics, Dept. of Physics and Astronomy Northwestern University, Evanston, USA

ABSTRACT

Computing the interactions between the stars within dense stellar clusters is a problem of fundamental importance in theoretical astrophysics. However, simulating realistic sized clusters of about 10^6 stars is computationally intensive and often takes a long time to complete. This paper presents the parallelization of a Monte Carlo method-based algorithm for simulating stellar cluster evolution on programmable Graphics Processing Units (GPUs). The kernels of this algorithm involve numerical methods of root-bisection and von Neumann rejection. Our experiments show that although these kernels exhibit data dependent decision making and unavoidable non-contiguous memory accesses, the GPU can still deliver substantial near-linear speed-ups which is unlikely to be achieved on a CPU-based system. For problem sizes ranging from 10^6 to 7×10^6 stars, we obtain up to $28\times$ speedups for these kernels, and a $2\times$ overall application speedup on an NVIDIA GTX280 GPU over the sequential version run on an AMD© Phenom™ Quad-Core Processor.

General Terms

PERFORMANCE

Keywords

Graphics processing unit (GPU), CUDA, Monte Carlo simulation, bisection method, parallel random number generator, multi-scale simulation.

1. INTRODUCTION

The dynamical evolution of dense star clusters is a problem of fundamental importance in theoretical astrophysics, but many aspects of it have remained unresolved in spite of years of numerical work and improved observational data [12]. Important examples of star clusters include *globular clusters*, spherical systems containing typically 10^5 - 10^7 stars densely packed within radii of just a few light years, and *galactic nuclei*, even denser systems with up to 10^9 stars contained in similarly small volumes, and often surrounding a supermassive black hole at the center. These are collisional, or relaxation dominated systems, where relaxation is the collective effect of many weak, random gravitational encounters between stars, as opposed to collision-less systems like galaxies. Studying their evolution is critical to many key unsolved problems in astrophysics. It connects directly to our understanding of star formation, as well as galaxy and supermassive black hole formation and evolution. Dynamical interactions in dense star clusters play a key role in the formation of many of the most interesting and exotic astronomical sources, such as bright X-ray and gamma-ray sources, radio pulsars, and supernovae.

The evolution of dense star clusters is a challenging multi-physics, multi-scale problem, consisting of three key elements: (i) *stellar dynamics*, which is relevant on timescales from 10^{-4} to 10^{10} years and on spatial scales from 10^{11} to 10^{19} cm; (ii) *single and binary star evolution*, with timescales from 10^3 to 10^{10} years and spatial scales from 10^6 to 10^{13} cm; and (iii) *hydrodynamics of close stellar interactions*, relevant on timescales from 10^{-4} to 1 year and spatial scales of 10^{10} to 10^{13} cm. The primary challenge lies in the tight coupling of these scales and physical processes as they influence one another both locally through close encounters or collisions and globally through long-range, gravitational interactions of the whole system.

There are two main approaches to simulating the gravitational dynamics of dense stellar systems. The direct N-body method evolves each point-particle by summing up all inter-particle forces, and thus requires N^2 force calculations per timestep [1, 16], where N is the number of particles. The orbit-averaged technique solves for the evolution of stars in energy and angular momentum space, allowing for a scaling closer to $O(N \log N)$, but assumptions of spherical symmetry and dynamical equilibrium have to be made.

Although N-body simulations have been performed for a large number of stars (eg. [10]) for collision-less systems, the methods employed are unsuitable for dense stellar systems

978-1-4673-2633-9/12/$31.00 ©2012 IEEE

where the evolution is dominated by relaxation. The inclusion of all relevant physics for collisional systems and the steep N^2 scaling have limited N-body simulations of dense stellar systems to $N \lesssim 10^5$. However, the typical number of stars in globular clusters and galactic nuclei is 1 to 5 orders of magnitude larger than that. For example, to date, only Hurley [15] carried out direct N-body simulations of globular clusters containing 10^5 stars with up to 10% primordial binaries. Even with this limited number of stars, each of these simulations took half a year to complete on special purpose hardware (GRAPE processors [6]), obviously limiting assessment of the robustness of results and preventing any kind of extensive parameter-space exploration. Although recent advances in high performance computing have led to the development of a number of parallel N-body codes [7, 8, 33], there does not exist one that includes all relevant physics and can still simulate a realistic number of stars.

Recently, the Cluster Monte Carlo (CMC) algorithm [2, 5, 17, 18, 32] which is based on the orbit-averaging method, demonstrated that it can simulate the evolution of systems containing up to a few million stars. Yet, a typical simulation of about a million stars up to average cluster ages of 10 billion years takes typically 3 - 4 weeks on a modern desktop computer, which means simulations of clusters of 10^7 stars will take a prohibitive amount of time. Moreover, there has been no previous parallel implementation of the algorithm. To accelerate simulations of such large data sizes, High Performance Computing (HPC) techniques can play a key role. Many special accelerator technologies and many-core architectures such as FPGAs [27, 31], and GPUs [4, 31] can be of great help for such intensive computations. With General Purpose Graphics Processing Units (GPGPUs) becoming increasingly powerful, inexpensive, and relatively easy to program, they have become a very attractive hardware acceleration platform.

In this paper, we present a GPU accelerated implementation of the CMC algorithm. The contributions of this paper are as follows: 1) We present a GPU accelerated CMC algorithm for dense stellar systems which allows us to explore physical regimes which were out of reach of current astrophysical simulations. 2) We devise a memory-layout strategy to pack the data elements before transferring them to the GPU which optimizes the global memory reads and writes. This idea can be reused for parallelizing applications having similar data structures. 3) We present a parallel random number generator which uses a splitting method to generate parallel statistically independent random numbers. We also discuss the problems related to consistency of results that arise, and a possible way to resolve them that could be employed in other parallel Monte Carlo algorithms. 4) Our experimental results show that for memory-access bound algorithms, the GPU, owing to its much higher memory bandwidth is a much better platform for parallelization compared to multicore architectures.

The rest of the paper is organized as follows: In Section 2, we present an overview of the CMC algorithm, its complexity, and performance bottlenecks. Section 3 talks about the design and implementation of our GPU accelerated version. Finally, in Section 4, we discuss experimental results.

2. OVERVIEW OF CMC

The CMC algorithm is developed to model the evolution of massive globular clusters and galactic nuclei with a realistic number of stars, and at the same time account for all relevant physical processes including treatment of strong binary interactions, physical stellar collisions, and stellar evolution. Its results agree very well with more accurate but much slower methods such as N-body and direct numerical integrations of the Fokker-Planck equation [2, 5, 17, 32]. The algorithm calculates the overall evolution of the cluster over many timesteps by keeping track of the physical properties (such as mass, orbital energy, angular momentum, etc.) of individual stars. The cluster may contain different types of systems (single stars, binaries, etc.) and their properties can change over time based on two-body relaxation, stellar collisions and binary interactions. The CMC algorithm handles these multi-physics by running different algorithms for different types of physical systems and processes.

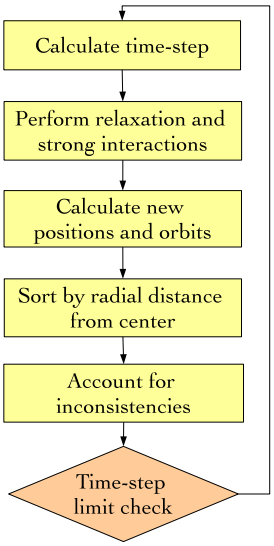

Figure 1: Flowchart of the CMC algorithm.

Figure 1 shows the flowchart of the CMC algorithm. The outer loop keeps track of the total number of simulation timesteps and terminates when a user-specified limit is reached. It consists of the following computational kernels. (1) **Timestep calculation** - Due to the multi-scale nature of the problem, different physical processes use different time scales. Moving averages are used to calculate the timesteps for each process. The smallest among them is chosen to be the global timestep shared by all processes during the iteration. This ensures no interesting process is missed at any timestep during the simulation. (2) **Relaxation and strong interactions** - Three algorithms can be chosen to run based on the physical system type. (i) *Two-body relaxation* evaluates an analytic expression for a representative encounter between two nearest-neighbors. (ii) *Binary interactions* does a direct integration of Newton's equations for 3 or 4 bodies using the 8^{th} order Runge-Kutta Prince-Dormand method. (iii) *Stellar collisions* merges two bodies based on the local collision probability and changes their properties correspondingly. (3) **New orbits computation** - The processes in (2) cause changes in the orbital properties of the stars and thus their probability to be found at a given location at the next timestep in the cluster. According to this probability distribution, the kernel samples new positions. It uses the

bisection method to calculate the extension of the new orbits and von Neumann rejection sampling to sample the new positions. (4) **Sort stars by radial distance** - This kernel sorts the stars based on their new radial distances. Sorting the stars is essential to determine the nearest neighbors for relaxation and strong interactions, and for computing the gravitational potential at various radial distances from the center (discussed in more detail in section 2.2). (5) **Account for inconsistencies** - As the new orbits of the stars were calculated in the gravitational potential of the previous timestep, this kernel accounts for this time-lag in potential in order to make sure conservation laws are obeyed.

2.1 Complexity and Performance Analysis

We now analyze the time complexity of these kernels per timestep. The '*timestep calculation*' kernel involves the computation of moving averages for each star over a window of 10 stars on either side, and hence takes $O(N)$ time, where N is the number of stars. The '*relaxation and strong interactions*', and '*account for inconsistencies*' kernels contains a set of $O(1)$ calculations inside a loop over all stars. Hence, these two kernels have the time complexity of $O(N)$. To determine the new orbits for each star, the third kernel finds the roots of an expression on an unstructured one-dimensional grid through bisection. This uses the bisection method which has a time complexity of $O(N \log N)$. The radial sorting of all stars uses the Quick Sort algorithm and has the same time complexity.

To identify any potential performance bottlenecks, we ran a simulation of 2 million stars on a 2.6 GHz AMD© Phenom™ Quad-Core Processor with 8 GB of RAM and profiled the kernels. Table 1 shows the time taken by each kernel per timestep, for a simulation up to 10^5 timesteps, and the percentage each kernel contributes to the total execution time. We can see that the '*new orbits calculation*' kernel is a clear bottleneck, and consumes the majority of the total execution time. For a simulation run up to 10^5 timesteps (typical for globular cluster simulations), this kernel would take ~ 17 among the ~ 31 days the entire simulation would take. Reducing the time taken by this kernel will therefore result in significant performance improvement of the CMC algorithm as a whole.

2.2 Bottlenecks

The '*new orbit calculation*' kernel consists of two parts. The first finds the pericenter and apocenter distances of a star's new orbit. This is done by calculating the roots of a function using the bisection method. The second finds the star's new position in the newly generated orbit using von Neumann rejection sampling. Hénon [13] describes the orbit sampling procedure in more detail.

The algorithm computes the gravitational potential of the cluster by summing the potential due to each star, under the assumption of spherical symmetry. It maintains only the radial position r of each star (due to the spherical symmetry assumption, the angular positions can be neglected to a very good approximation in globular clusters and galactic nuclei). After the stars are sorted by increasing radii, the potential at a point r, which lies between two stars at positions r_k and r_{k+1}, is given by

$$\Phi(r) = G\left(-\frac{1}{r}\sum_{i=1}^{k} m_i - \sum_{i=k+1}^{N}\frac{m_i}{r_i}\right) \tag{1}$$

where m_i is the mass, r_i the position of star i, and G the gravitational constant.

At the beginning of every timestep, the algorithm computes and stores the potential $\Phi_k = \Phi(r_k)$ at radial distances r_k $(k = 1, ..., N)$ which are the positions of the stars. To get the potential $\Phi(r)$ at any radial position r from the center, one first has to find the k such that $r_k \le r \le r_{k+1}$ and then compute $\Phi(r)$ by:

$$\Phi(r) = \Phi_k + \frac{1/r_k - 1/r}{1/r_k - 1/r_{k+1}}(\Phi_{k+1} - \Phi_k) \tag{2}$$

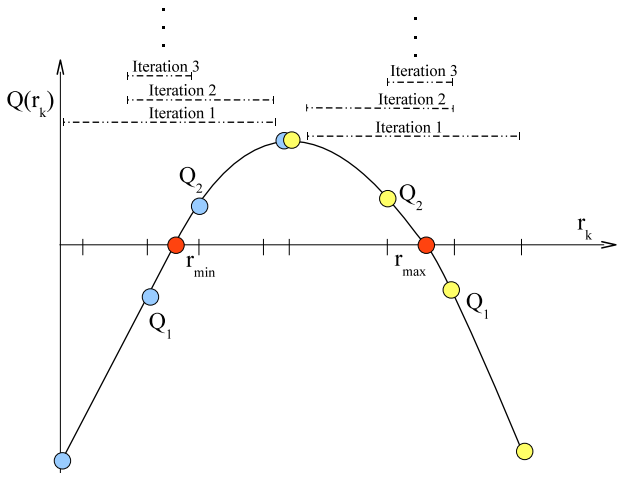

Figure 2: Iterations of the bisection method converging on the roots $r_{\min}$ and $r_{\max}$.

Given a star with energy E and angular momentum J moving in the gravitational potential $\Phi(r)$, its rosette orbit r oscillates between two extreme values $r_{\min}$ and $r_{\max}$, and are roots of:

$$Q(r) = 2E - 2\Phi(r) - J^2/r^2 = 0 \tag{3}$$

Since the potential at an arbitrary point cannot be analytically calculated in our case, the interval in which $r_{\min}$ and $r_{\max}$ falls must be first found. This is done by determining k such that $Q(r_k) < 0 < Q(r_{k+1})$. We use the bisection method shown in Algorithm 1 to do this. It starts with two values of k, k_{left} and k_{right}, and at each step divides the interval into two parts, retaining only the one in which the solution is contained and discarding the other. Figure 2 shows the $Q(r_k)$ function and the iterations of bisection method converging onto the roots. Once the interval is found, Φ, and thus Q, can be computed analytically in that interval, and determining $r_{\min}$ and $r_{\max}$ become a simple arithmetic operation. The bisection method is a divide-and-conquer algorithm and has a complexity of $O(N \log N)$, and since this is done for all N stars, the total run-time is $O(N \log N)$.

The next step is to select a position of the star in the new orbit between $r_{\max}$ and $r_{\min}$. The probability to choose it in an interval dr should be equal to the fraction of time spent by the star in dr, i.e.:

$$\frac{dt}{T} = \frac{dr/|v_r|}{\int_{r_{\min}}^{r_{\max}} dr/|v_r|} \tag{4}$$

with the radial velocity $v_r = [Q(r)]^{1/2}$.

Kernel	Time per timestep (sec)	Time for 10^5 timesteps (days)	% Time
Timestep calculation	2.39	2.77	9%
Relaxation and strong interactions	3.33	3.85	13%
New orbits calculation	14.59	16.89	54%
Sorting by radial distance	4.11	4.76	15%
Account for inconsistencies	2.26	2.62	9%
Total	26.68	30.89	100%

Table 1: Execution time break-up for various kernels of the CMC algorithm.

Algorithm 1 Bisection algorithm

```
if Q(k_min) ≤ Q(k_max) then
    while k_max ≠ k_min do
        k_try = (k_min + k_max + 1)/2
        if Q(k_try) < 0 then
            k_min = k_try
        else
            k_max = k_try − 1
        end if
    end while
    while k_max ≠ k_min do
        k_try = (k_min + k_max + 1)/2
        if Q(k_try) > 0 then
            k_min = k_try
        else
            k_max = k_try − 1
        end if
    end while
end ifreturn k_min
```

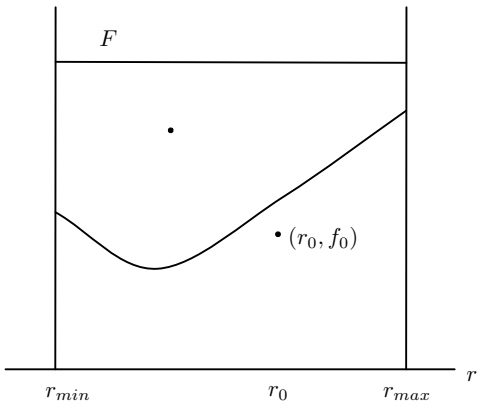

Figure 3: The rejection technique showing functions $f(r)$ and F, the rejected point and the accepted point (r_0, f_0) [13].

The computation of the half-period T is done by the classical von Neumann rejection technique [11]. We want a probability distribution to a known function $f(r)$, without knowing the constant of proportionality. We take a number F which is everywhere larger than $f(r)$ (refer Fig 3) and select a point (r_0, f_0) at random in the rectangle $r_{\min} < r_0 < r_{\max}$ and $0 < f_0 < F$, with a uniform distribution. In other words, we compute:

$$r_0 = r_{\min} + (r_{\max} - r_{\min})X \tag{5}$$

$$f_0 = FX' \tag{6}$$

where X and X' are a pair of normalized random numbers. If the point is below the curve: $f_0 < f(r_0)$, we take $r = r_0$ as the selected value. In the opposite case, this value is rejected and a new point in the rectangle is tried with a fresh pair of random numbers. This process continues until a point below the curve is obtained. In the CMC algorithm a slightly modified version of the method is used, since $f(r) = 1/|v_r|$ becomes infinite at both ends of the interval. However, the same principle is used.

The complexity of this part is dominated by the fact that the potential has to be calculated at the random point chosen to check whether the point should be accepted or rejected. Since the potential is not available at any random point chosen, this requires another root-bisection, which makes the complexity of this step $O(N \log N)$.

3. DESIGN AND IMPLEMENTATION

Based on our experimental profiling and complexity analysis, the '*new orbits calculation*' kernel is the bottleneck consuming the majority (54%) of the total run-time with a complexity of $O(N \log N)$. We accelerate this portion using a GPU. The CMC algorithm is programmed in C, and we used CUDA for the GPU implementation.

We assign one thread on the GPU to do the computations for one star. This ensures minimal data dependency between the threads since the same set of operations are performed on different data, and makes the bisection method and rejection technique implementations naturally suited for SIMD (Single Instruction, Multiple Data) architectures.

3.1 Memory Access Optimization

To harness the high computation power of the GPU, it is very essential to have a good understanding of its memory hierarchy in order to develop strategies that reduce memory access latency. The first step towards optimizing memory accesses is to ensure that memory transfer between the host and the GPU is kept to a minimum. Another important factor that needs to be considered is global memory coalescing in the GPU which could cause a great difference in performance. When a GPU kernel accesses global memory, all threads in groups of a *half-warp* [28] access a bank of memory at the same time. Coalescing of memory accesses happens when data requested by these groups of threads are located in contiguous memory addresses, in which case they can be read in one (or very few number of) access(es).

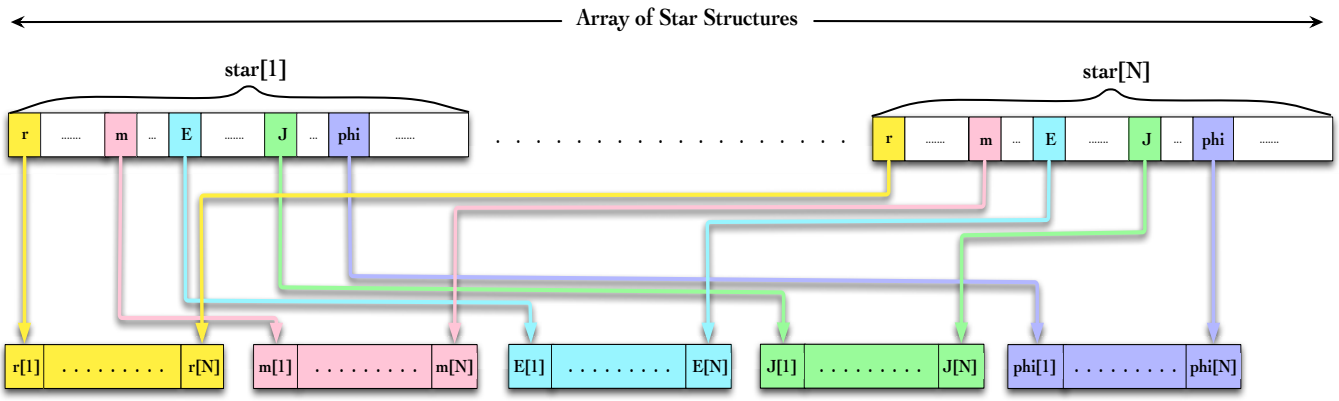

Figure 4: Data coalescing strategy used to strip the original star data structure and pack into contiguous arrays before transferring them to the GPU.

Hence, whether data is coalesced or not has a significant impact on an application's performance as it determines the degree of memory access parallelism.

In CMC, the physical properties of each star are stored in a C structure, containing 46 double precision variables. The N stars are stored in an array of such C structures. Figure 4 shows the original data layout on top i.e. an array of structures. The kernels we parallelize only require 5 among the 49 variables present in the star structure: radial distance (r), mass (m), energy (E), angular momentum (J), and potential at r (phi) which are shown in color.

To achieve coalesced memory accesses, we need to pack the data before transferring it to the GPU in a way that they would be stored in contiguous memory locations in the GPU global memory. A number of memory accesses involve the same set of properties for different stars being accessed together by these kernels since one thread works on the data of one star. Hence, we extract and pack these into separate, contiguous arrays, one for each property. This ensures that the memory accesses in the GPU will be mostly coalesced. Also, by extracting and packing only the 5 properties required by the parallel kernels, we minimize the data transfer between the CPU and GPU. Figure 4 gives a schematic representation of this data reorganization.

3.2 Random Number Generation

For Monte Carlo simulations, generation of random numbers is very essential [14, 22–24]. Typically, this is done by the use of a pseudo random numbers generator (PRNG). PRNGs maintain a set of states which are used to calculate the random number. The values of these states are updated to new ones every time a random number is generated. A PRNG can be started from an arbitrary starting state using a random seed. It will always produce the same sequence thereafter when initialized with the same seed. The maximum length of the sequence before it begins to repeat is called the period of the PRNG. In the CMC algorithm, we use a combined Tausworthe PRNG [9] which is a maximally equidistributed combined linear feedback shift register generator [20,21] whose period is $\sim 2^{113}$. We use this generator due to its property of maximal equidistribution, as it ensures that the random numbers are distributed uniformly and the sequence does not have gaps in resolution [20, 21] which is especially important for the very low probabilities we sample for, e.g., collisions between single stars. The lack of this property in some PRNGs prevents us from using libraries that support parallel random number generation such as CURAND [29].

In order to implement a parallel version of the rejection technique, we would like each thread to maintain its own independent state information and generate random numbers as and when required. Yet, we also require these concurrently generated random numbers to maintain statistical independence. In short, we need a number of statistically independent PRNGs, one for each thread. An intuitive idea to do this would be to allocate separate state information for each instance of the PRNG and make sure each PRNG modifies only its own state. However, this does not assure statistical independence and can suffer from serious correlation problems. Any PRNG generates a single output sequence, and seeding the PRNG chooses a particular starting place in the sequence. Two instances are considered to be independent if their sequences do not overlap during execution of the program. In our case even if we initialize two PRNGs with different seeds, they will start with some starting place in the same sequence which will overlap eventually.

A very efficient technique to produce multiple states from a single random sequence is to use jump functions. We follow a procedure [3] that would generate multiple states from a single seed by repeatedly applying the jump functions and saving intermediate results. We implemented this task on the host (CPU), and it performs extremely fast (in the order of microseconds). The jump displacement (number of random numbers between two states) depends on the number of random numbers required by each thread for the entire simulation. In our implementation, we chose a displacement of 2^{80} to ensure a sufficient number of random numbers per thread while still allowing a large number of threads to run in parallel. Once generated on the CPU, these states are transferred to the GPU global memory. Each thread reads the respective starting state from the memory and produces random numbers independently.

However, in the GPU-accelerated version, the way random numbers are assigned to stars is different from the original serial version. This brings in a problem of inconsistency in the results of a serial and parallel simulation, and verifying

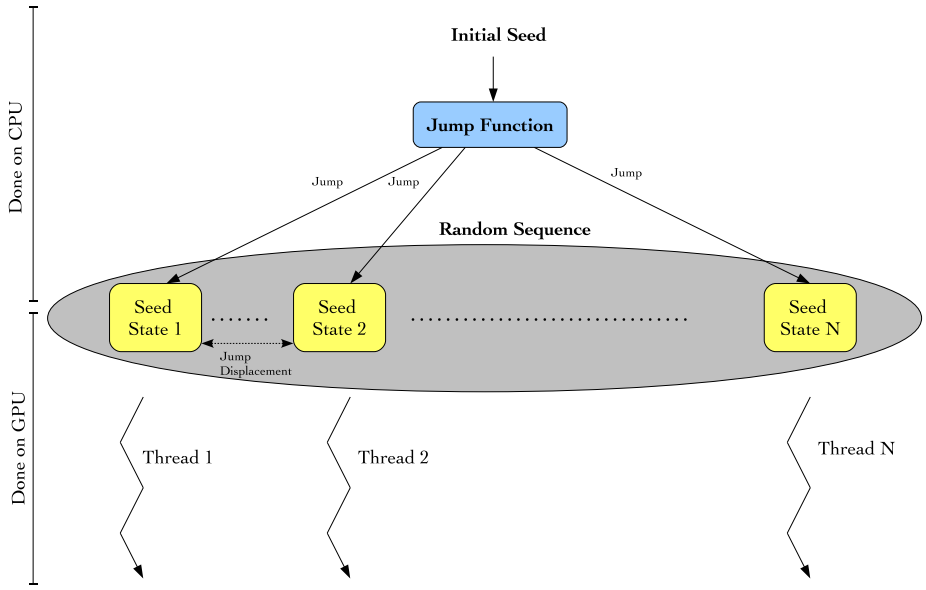

Figure 5: Technique used to generate multiple statistically independent random states. This is done on the CPU, after which the states are transferred to the GPU to allow the thread to use them to generate random numbers independently.

the correctness of results of a parallel version becomes a problem. In order to verify the correctness of the results, we change the serial version such that the same mapping of random numbers to particles is followed by the serial version as used in the parallel version. For our choice of random number generator this does not change in any significant way the statistical properties of the generated random numbers.

An implementation of the parallel Tausworthe 113 random number generator is publicly available at `http://ciera.northwestern.edu/Research/CS_Astro.php`

3.3 Limitations

In spite of our efforts to optimize the kernel implementation, there are two factors that prevent us from harnessing the complete potential of the GPU. First, in the bisection kernel, the algorithm starts with two end points of an array, and at each step computes their mid-point. It then picks the interval that contains the solution, and repeats the process with the new end points, i.e. the previously computed mid-point and one of earlier end points. These mid and end points are separated by a large distance in most cases. Therefore, the memory reads of these array elements are not contiguous. The only exception is towards the end of the bisection, when it has almost converged on to the root. At this time these points will be close together and hence the memory reads might be coalesced. As these accesses (mid-points) are data-dependent and cannot be easily predetermined, non-coalesced accesses cannot be avoided. Such random accesses leaves us with no simple way of using other types of faster memory on the GPU memory hierarchy such as shared and texture memory.

Secondly, in the rejection kernel, random numbers are repeatedly generated until the random number satisfies a specified termination condition. Since the random numbers used by the threads are completely independent, different threads may fulfill the condition and terminate the loop at different times during the execution. In this case, the terminated threads will remain idle until all the other threads complete. Such idling of threads in a GPU warp can significantly impede the performance. We minimize such branch divergence by running the loop for a fixed, large number of iterations and making all threads perform redundant calculations after the termination condition is satisfied. As it turns out, however, the overhead due to branch divergence was very small, and we did not obtain a measurable performance gain. On the other hand, it could be an idea that could be applied to solve other similar problems.

4. EXPERIMENTS AND RESULTS

All our experiments are carried out on a 2.6 GHz AMD© Phenom^{TM} Quad-Core Processor with 2 GB of RAM per core running Fedora 9 Linux, and an NVIDIA GTX280 GPU with 30 multiprocessors, and 1 GB of RAM, using the version 3.1 of the CUDA compiler. All computations are done in double precision, and the GTX280 GPU has only a single Double Precision Unit (DPU) per multiprocessor.

4.1 Performance Analysis

We collect the timing results for 5 simulation timesteps of a single-mass cluster with a Plummer density profile [30], and sizes ranging from 10^6 to 7×10^6 stars, encompassing nearly all globular cluster sizes (see, e.g., [25]). Figure 6 compares the GPU and CPU run-times. Figure 7 shows the speedup of '*new orbits calculation*' part and the bisection and rejection kernels individually. We see that the average speedups for the rejection and bisection kernels are 22 and 31, respectively. This is due to the difference in the number

of floating point operations between the two kernels which is a factor of 10. This makes a major difference on the CPU but not on the GPU as it has more ALUs, and hence the difference in speedup. We also observe that the total speedup increases slightly as the data size increases. In general, we obtain a very good scalability. A quick calculation on the execution times and data sizes in Figure 6 shows that the GPU scalability strictly follows the kernel's complexity $O(N \log N)$. Note that as the memory transfer between the CPU and GPU is currently not optimized, our speedup calculations do not include that overhead. However, as we transfer only a subset of the entire data for each star, there is the potential space for improvement to interleave kernel computations with data transfer and substantially reduce this overhead.

In Figure 8, the run-times for the bisection and rejection kernels on the GPU are shown. Both kernels take about the equal amount of time in total on the GPU for all data sizes. This indicates that the performance of these kernels is limited by the memory bandwidth as they roughly require the same amount of global memory accesses.

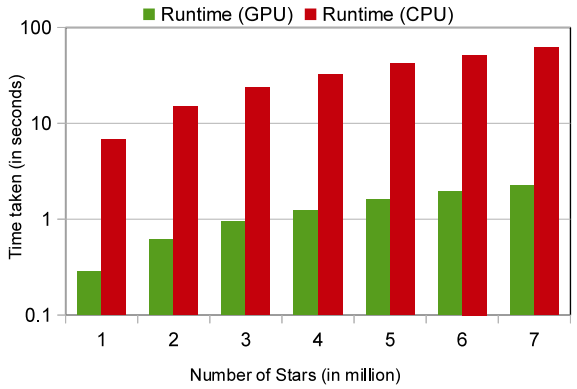

Figure 6: Comparison of total run-times of the sequential and parallelized kernels for various data sizes.

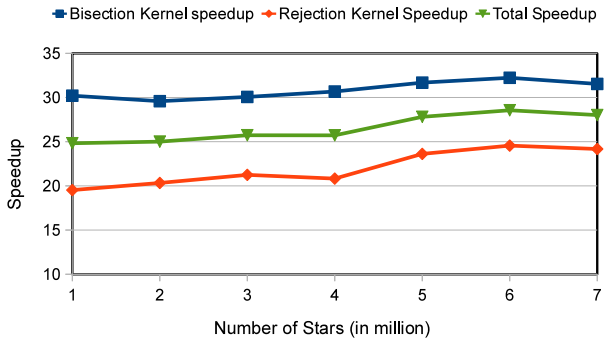

Figure 7: Total speedup, and speedups of the bisection and rejection kernels.

We also evaluated our implementation with different physical configurations. With the same range of data sizes, we simulate clusters with two different density profiles - Plummer [30] and King [19], whose distribution functions have

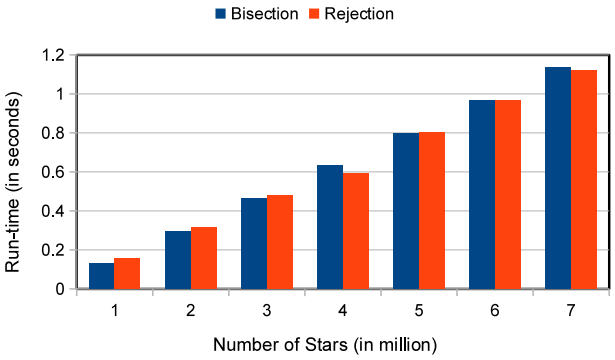

Figure 8: Run-times of the bisection and rejection kernels on the GPU for various data sizes.

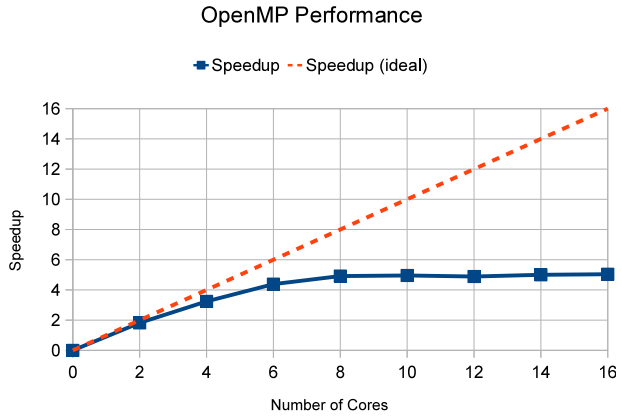

Figure 9: Speedup obtained from a shared memory implementation of the Bisection kernel for a cluster with 1 million stars run on a 16-core shared memory machine.

different forms. Figure 10 shows the comparison of run-times for these two configurations. We can see that the run-times for both configurations are almost the same, from which we infer that the physical conditions of the clusters do not influence the performance of our kernels. This is due to the fact that changes in physical configurations do not affect the computations in our kernels in any significant way, resulting in equally good performance for all configurations.

4.2 GPU vs. Multi-core Architectures

For comparison, we also implemented the bisection kernel using OpenMP with some optimizations to improve data locality. We ran a simulation of 1 million stars with the same density profile as before on a 16-core shared memory machine with 4 Quad-Core AMD Opteron^{TM} Processors. Figure 9 shows the speedup we obtained compared to the ideal speedup. We can see that the speedup increases sub-linearly till 8 cores after which it saturates. This is possibly due to memory contention, bandwidth limitations and/or effects due to Non-Uniform Memory Access (NUMA) [26]. Although we reserve a thorough investigation of this for a future work, with the given data it appears to be unlikely that even larger shared memory machines can deliver speedups as high as a GPU for these kernels.

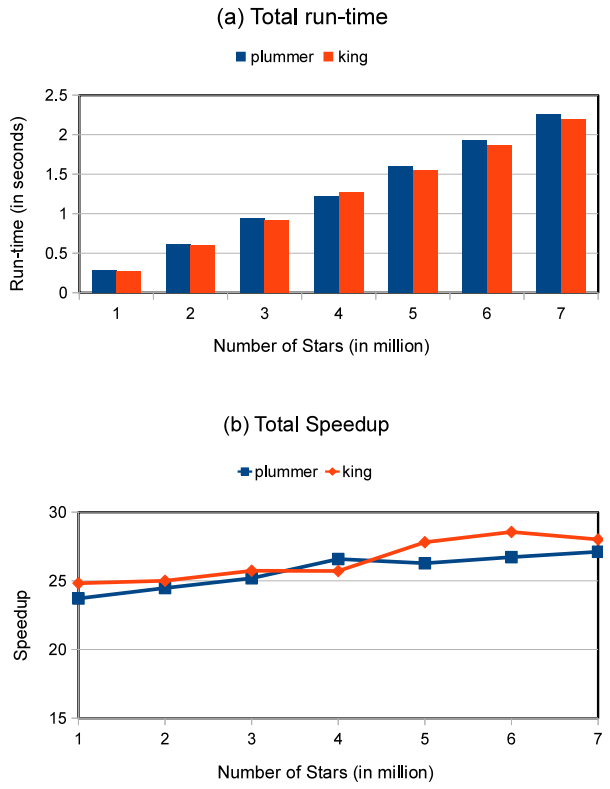

Figure 10: Comparison of total run-times and speedups for the Plummer and King density profiles for the parallelized kernels.

4.3 Data and Workload Partitioning

We partitioned our data space into one-dimensional grid of blocks on the GPU. Due to the complexity of the expressions involved in the calculations of orbital positions, our kernels use a significant amount of registers (64 registers per thread). Thus, the block dimension is restricted to 256 threads per block as the GTX280 GPU has only 16384 registers per block. To analyze the performance, we first made a parameter scan in the block and grid dimensions by varying the block sizes from 64 to 256 and the grid sizes from 12 to 72. Figure 11 shows the plot of the execution time as a function of the total number of threads. The time decreases with increasing number of threads and saturates at about 6000 threads. This clearly shows that the run-time mainly depends on the total number of threads. The decrease in run-time is expected as the GPU utilization is increased for larger numbers of threads. The saturation is due to either the limited global memory bandwidth, or the limited number of multiprocessors. The NVIDIA GTX280 GPU has 30 multiprocessors, and when we multiply the maximum block dimension 256 by 30, this number (7680) is close to the saturation point. This strongly suggests an effect of the number of multiprocessors on the saturation.

Figure 12 shows the variation of run-time with the grid dimension for constant number of threads (5760). To ensure no performance loss due to the block-size not being multiples of the half-warp size (16 for the GTX280 GPU), we choose the block sizes of multiples of 16. We observe that the performance can degrade by 50% from the best case for poor choices of grid dimensions. The best case is achieved when the number of blocks is a multiple of the number of multiprocessors, as it ensures full GPU utilization.

We also study the likely influence of effective memory bandwidth. Figure 13 shows a plot of the run-time with varying number of threads for a fixed grid dimension, 60, a multiple of the number of multiprocessors. We expect a linear decrease with increase in the number of threads per block, but as we can see this is not the case. Thus, it is plausible that this indicates the performance being limited by the memory bandwidth. Another factor that could limit our performance is branch divergence the effect of which is more difficult to measure. However, we plan to quantify this effect in a future study.

5. CONCLUSION AND FUTURE WORK

In this paper, we eliminate the most compute-intensive bottlenecks of the CMC algorithm by implementing them on a GPU. For cluster sizes ranging from 10^6 to 7×10^6 stars, we obtain a mean speedup of 28× for these kernels. Our implementation also has very good scalability with the data size and performs equally well on all physical configuration of clusters simulated. The bottlenecks originally took $\sim 54\%$ of the total runtime, and by Amdahl's Law, this acceleration delivers a $\sim 2\times$ speedup of the entire simulation.

We present strategies to handle branch divergence, and to reorganize the data structure to optimize memory reads and writes on the GPU. In addition, we show how a good choice of data partitioning on the GPU could make a significant difference in the performance. To produce parallel statistically independent random numbers, we use a splitting method which splits the output of our PRNG into a number of substreams. The substreams are then used by different threads of the GPU to generate random numbers. Although generation of statistically independent random numbers is critical for Monte Carlo simulations, this might cause a difference in the results of the serial and parallel versions. We handle this by letting the serial version emulate the random number generation and data partitioning of the parallel version which then guarantees the output of the two versions to be identical except for the least significant bit. We observe that the parts we parallelize on the GPU are limited by memory-bandwidth rather than computations, and hence the performance of our implementation is not comparable to the theoretical peak performance of the NVIDIA GTX280 GPU. However, we show that the GPU is still a much better platform for parallelization compared to multi-core architectures due to its relatively higher memory bandwidth.

The GPU-accelerated version significantly reduces the run-time of the CMC algorithm, and enables simulations of large stellar clusters which were previously very time-consuming. In this work, we accelerated only the '*new orbits calculation*' kernel using a GPU. The other kernels exhibit a much higher data dependency and conditional branching, and therefore not ideal for a GPU. Also, with the current version, simulations of clusters beyond 10^7 stars exceed the memory capacity of a single GPU. This means we cannot simulate galactic nuclei which can have up to 10^9 stars, a typical problem size interesting to the computational astrophysics community. We plan to develop an MPI-OpenMP-CUDA hybrid approach to parallelize the entire CMC algorithm on large-scale high-performance computers, in which we will use MPI

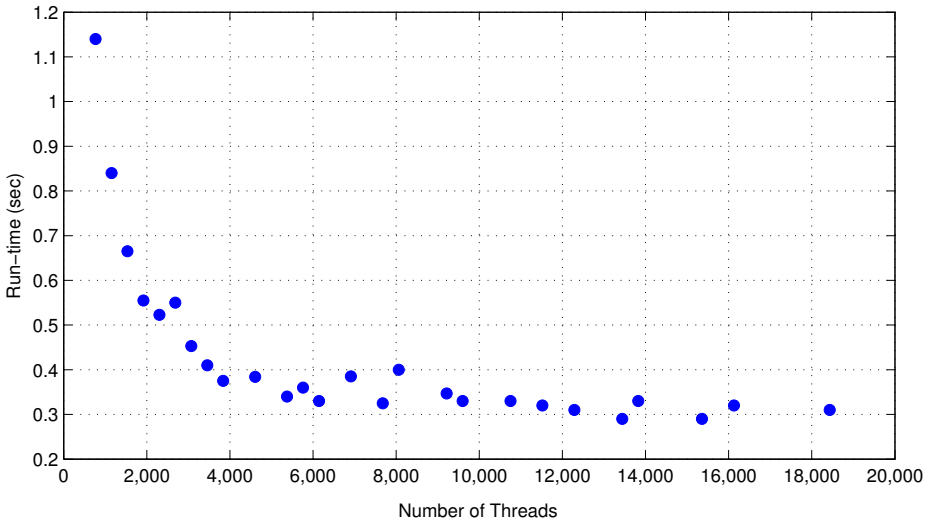

Figure 11: Dependence of the total run-time of our kernels on the total number of threads.

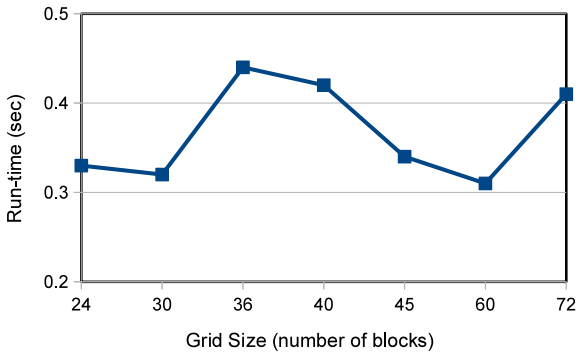

Figure 12: Variation of run-time with grid size for a fixed value of total number of threads (5760).

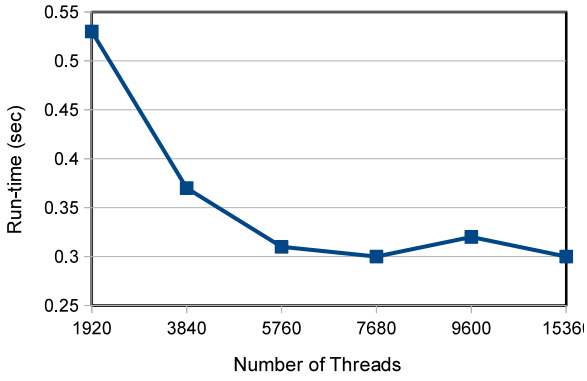

Figure 13: Variation of run-time with total number of threads for a fixed grid dimension (60).

for inter-node communication, OpenMP for shared-memory multi-core CPUs, and CUDA for GPUs.

6. ACKNOWLEDGEMENTS

This work is supported by NSF Grant AST-0607498 at Northwestern University, and in part by the National Science Foundation under award numbers PHY05-51164, CCF-0621443, OCI-0724599, CCF-0833131, CNS-0830927, IIS-0905205, OCI-0956311, CCF-0938000, CCF-1043085, CCF-1029166, and OCI-1144061, and in part by DOE grants DE-FC02-07ER25808, DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309, and DE-SC0005340.

7. REFERENCES

[1] S. J. Aarseth. *Gravitational N-Body Simulations*. Cambridge University Press, 2003.

[2] S. Chatterjee, J. M. Fregeau, S. Umbreit, and F. A. Rasio. Monte Carlo simulations of globular cluster evolution. V. Binary stellar evolution. *The Astrophysical Journal*, 719(1):915, 2010.

[3] J. C. Collins. Testing, selection, and implementation of random number generators. *Army Research Laboratory*, pages 4, 41, 2008.

[4] C. J. Fluke, D. G. Barnes, B. R. Barsdell, and A. H. Hassan. Astrophysical supercomputing with GPUs: Critical decisions for early adopters. *ArXiv e-prints*, Aug. 2010, astro-ph.IM/1008.4623.

[5] J. M. Fregeau and F. A. Rasio. Monte Carlo simulations of globular cluster evolution. IV. Direct integration of strong interactions. *The Astrophysical Journal*, 658(2):1047, 2007.

[6] T. Fukushige, J. Makino, and A. Kawai. GRAPE-6A: A single-card GRAPE-6 for parallel PC-GRAPE cluster system. *Publications of the Astronomical Society of Japan*, 57(6):1009–1021, 2005, arXiv:astro-ph/0504407v1.

[7] E. Gaburov, J. Bédorf, and S. P. Zwart. Gravitational tree-code on graphics processing units: implementation in CUDA. *Procedia Computer Science*, 1(1):9, 2010, arXiv:1005.5384v1.

[8] E. Gaburov, S. Harfst, and S. P. Zwart. SAPPORO: A way to turn your graphics cards into a GRAPE-6. *New Astronomy*, 14(7):630–637, 2009.

[9] M. Gil, G. H. Gonnet, and W. P. Petersen. A repetition test for pseudo-random number generators. *Monte Carlo Methods and Applications*, 12(5):385–393, Nov. 2006.

[10] T. Hamada and K. Nitadori. 190 TFlops astrophysical N-body simulation on a cluster of GPUs. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–9, Nov. 2010.

[11] J. M. Hammersley and D. C. Handscomb. *Monte Carlo methods*. Methuen, London, 1964.

[12] D. Heggie and P. Hut. *The gravitational million-body problem*. Cambridge University Press, 2003.

[13] M. H. Hénon. The Monte Carlo Method (Papers appear in the Proceedings of IAU Colloquium No. 10 Gravitational N-Body Problem (ed. by Myron Lecar), R. Reidel Publ. Co., Dordrecht-Holland.). *Astrophysics and Space Science*, 14:151–167, Nov. 1971.

[14] L. Howes. *Efficient Random Number Generation and Application Using CUDA*. Addison-Wesley, 2007.

[15] J. R. Hurley. Ratios of star cluster core and half-mass radii: a cautionary note on intermediate-mass black holes in star clusters. *Monthly Notices of the Royal Astronomical Society*, 379:93–99, July 2007, arXiv/0705.0748.

[16] P. Hut. The starlab environment for dense stellar systems. In J. Makino & P. Hut, editor, *Astrophysical Supercomputing using Particle Simulations*, volume 208 of *IAU Symposium*, pages 331–+, 2003, arXiv:astro-ph/0204431.

[17] K. J. Joshi, C. P. Nave, and F. A. Rasio. Monte Carlo simulations of globular cluster evolution. II. Mass spectra, stellar evolution, and lifetimes in the galaxy. *The Astrophysical Journal*, 550(2):691, 2001.

[18] K. J. Joshi, F. A. Rasio, and S. P. Zwart. Monte carlo simulations of globular cluster evolution. I. method and test calculations. *The Astrophysical Journal*, 540(2):969, 2000.

[19] I. R. King. The structure of star clusters. III. Some simple dynamical models. *The Astrophysical Journal*, 71:64–+, Feb. 1966.

[20] P. L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Mathematics of Computation*, 65:203–213, Jan. 1996.

[21] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68:261–269, Jan. 1999.

[22] M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, 2000.

[23] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[24] M. Matsumoto, I. Wada, A. Kuramoto, and H. Ashihara. Common defects in initialization of pseudorandom number generators. *ACM Trans. Model. Comput. Simul.*, 17(4):15, 2007.

[25] D. E. McLaughlin and R. P. van der Marel. Resolved massive star clusters in the milky way and its satellites: Brightness profiles and a catalog of fundamental parameters. *Astrophysical Journal Supplement Series*, 161:304–360, Dec. 2005, arXiv:astro-ph/0605132.

[26] S. Moreaud and B. Goglin. Impact of NUMA effects on high-speed networking with multi-opteron machines. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 24–29, Anaheim, CA, USA, 2007. ACTA Press.

[27] N. Nakasato and T. Hamada. Astrophysical hydrodynamics simulations on a reconfigurable system. *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 0:279–280, 2005.

[28] NVIDIA. CUDA programming guide 3.1, June 2010.

[29] F. Panneton and P. L'ecuyer. On the xorshift random number generators. *ACM Trans. Model. Comput. Simul.*, 15:346–361, Oct. 2005.

[30] H. C. Plummer. On the problem of distribution in globular star clusters. *Monthly Notices of the Royal Astronomical Society*, 71:460–470, Mar. 1911.

[31] R. Spurzem, P. Berczik, G. Marcus, A. Kugel, G. Lienhart, I. Berentzen, R. MŁnner, R. Klessen, and R. Banerjee. Accelerating astrophysical particle simulations with programmable hardware. *Computer Science - Research and Development*, 23:231–239, 2009.

[32] S. Umbreit, J. M. Fregeau, and F. A. Rasio. Monte Carlo simulations of globular cluster evolution. VI. The influence of an intermediate mass black hole. *arXiv:astro-ph/0910.5293*, page 45, 2009.

[33] S. F. P. Zwart, R. G. Belleman, and P. M. Geldof. High-performance direct gravitational N-body simulations on graphics processing units. *New Astronomy*, 12(8):641–650, 2007.