Volume Title
*ASP Conference Series, Vol. **Volume Number***
Author
*© **Copyright Year** Astronomical Society of the Pacific*

GPU-Accelerated Monte Carlo Simulations of Dense Stellar Systems

Bharath Pattabiraman[1,2], Stefan Umbreit[1,3], Wei-keng Liao[1,2], Frederic Rasio[1,3], Vassiliki Kalogera[1,3], and Alok Choudhary[1,2]

[1]*Center for Interdisciplinary Exploration and Research in Astrophysics, Northwestern University, Evanston, USA*

[2]*Dept. of Electrical Engineering and Computer Science, Northwestern University, Evanston, USA*

[3]*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 acceleration of a Monte Carlo algorithm for simulating stellar cluster evolution using programmable Graphics Processing Units (GPUs). This acceleration allows to explore physical regimes which were out of reach of current simulations.

1. Introduction

The evolution of dense star clusters is a challenging multi-physics, multi-scale problem, involving a wide range of spatial and temporal scales over which various physical processes operate and their tight coupling. One of the methods to model them is direct N-body, but its computational cost has a time-complexity of $O(N^3)$ (Heggie & Hut 2003) which results in poor scalability. Thus, the commonly-seen problem sizes of the direct N-body simulations are in the order of 10^5 stars, almost an order of magnitude lower than for most globular clusters observed in our galaxy.

Recently, the Cluster Monte Carlo (CMC) algorithm Joshi et al. (2000), a much faster approach, demonstrated that it can simulate the evolution of globular clusters 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 corresponding to 30 times longer. To accelerate simulations of such large data sizes, High Performance Computing (HPC) techniques can play a key role. With General Purpose Graphics Processing Units (GPGPUs) becoming increasingly powerful, inexpensive, and relatively easy to program, it has become a very attractive hardware acceleration platform. In this paper, we present a GPU accelerated implementation of the CMC algorithm.

2. Overview of CMC

The CMC code calculates the overall evolution of the cluster over many time-steps by

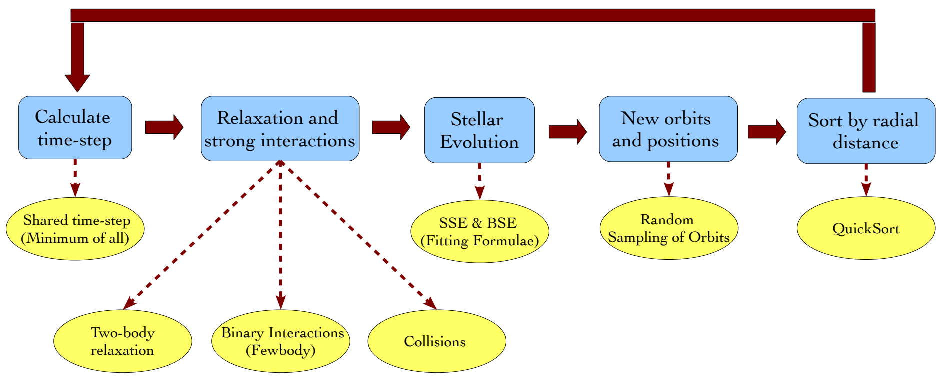
Calculate time-step
Relaxation and strong interactions
Stellar Evolution
New orbits and positions
Sort by radial distance
Shared time-step (Minimum of all)
SSE & BSE (Fitting Formulae)
Random Sampling of Orbits
QuickSort
Two-body relaxation
Binary Interactions (Fewbody)
Collisions

Figure 1. The CMC algorithm flowchart.

Figure 1 shows the flowchart of the CMC algorithm. It consists of the following kernels. (1) **Time-step calculation** - The smallest time-step of all processes is chosen to be the global time-step for the iteration. (2) **Relaxation and strong interactions** - where we do one of *two-body relaxation*, *binary interactions* or *stellar collisions* based on the physical system type. (3) **Stellar Evolution** - simulates the evolution of stars using the BSE and SSE codes (Hurley et al. 2002). (4) **New orbits computation** - samples new positions and orbits for all stars. (5) **Sort stars by radial distance** - sorts the stars based on their radial distances.

2.1. Performance Profiling

To identify any potential performance bottlenecks, we ran a simulation of 10^6 stars on a 2.6 GHz AMD© Phenom$^{\mathrm{TM}}$ Quad-Core Processor with 8 GB of RAM and profiled the kernels. Table 1 shows the percentage of time taken by each kernel per time-step.

Kernel	**% of time taken**
Relaxation and Strong Interaction	9%
Stellar Evolution	23%
New orbits calculation	53%
Sorting by radial distance	7%
Others	8%

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

We can see that the '*new orbits calculation*' kernel is the clear performance bottleneck. The following from Hénon (1971) describes the orbit sampling procedure in more detail. We compute the gravitational potential $U_k = U(r_k)$ at radial distances r_k $(k = 1, ..., N)$ which are the positions of the stars. To compute the gravitational potential $U(r)$ at any radial position r from the center, we first find the k such that $r_k \leq r \leq r_{k+1}$

and compute $U(r)$ using U_k:

$$U(r) = U_k + \frac{1/r_k - 1/r}{1/r_k - 1/r_{k+1}}(U_{k+1} - U_k) \tag{1}$$

Given a star with energy E and angular momentum J moving in a potential $U(r)$, its follows a rosette orbit with r oscillating between two extreme values r_{min} and r_{max}, which are roots of:

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

The interval in which r_{min} falls is found by looking up the sorted star list based on their radii and of the corresponding potentials U_k; that is, one determines k such that $Q(r_k) < 0 < Q(r_k + 1)$. We use the bisection method to do this root-finding which as a time complexity of $O(logN)$ and for all N stars, this becomes is $O(NlogN)$. Once the k values are found, computing r_{min} and r_{max} is simply an arithmetic operation.

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.:

$$\frac{dt}{T} = \frac{dr/\,|v_r|}{\int_{r_{min}}^{r_{max}} dr/\,|v_r|} \tag{3}$$

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

The computation of the half-period T is done by the classical von Neumann rejection technique (Hammersley & Handscomb 1964). 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 2) 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:

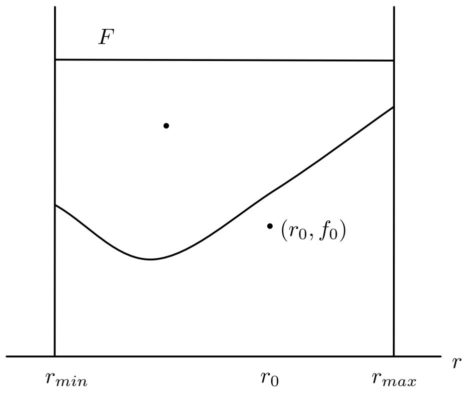

Figure 2. The rejection technique showing functions $f(r)$ and F, the rejected point and the accepted point (r_0, f_0) (Hénon 1971).

$$r_0 = r_{min} + (r_{max} - r_{min})X \tag{4}$$

$$f_0 = FX' \tag{5}$$

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, if not is rejected and repeated with a fresh point. This process continues until a point below the curve is obtained.

The orbit sampling part takes the maximum portion of the run-time and also has a time complexity of $O(NlogN)$ whereas the rest of the code scales as $O(N)$. Hence we decided to parallelize this on the GPU.

2.2. Implementation on the GPU

The bisection method and rejection technique are naturally suited for SIMD (Single Instruction, Multiple Data) architectures as there is minimal data dependency. However, to harness the high computation power of the GPU, it is important to achieve coalesced memory accesses. We pack the data before transferring it to the GPU in a way that data is stored in contiguous memory locations in the GPU global memory, and this tremendously improves the spe

We also need to produce multiple states from a single random sequence as this is essential to enable each thread generate its own random number, and at the same time maintain statistical independence between them. We use the technique mentioned in Collins (2008) to 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). 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 produce random numbers independently.

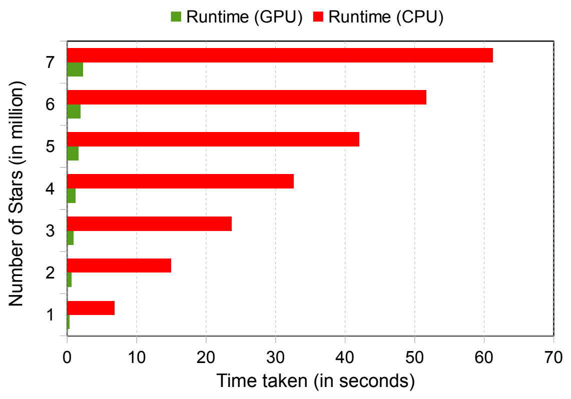

Figure 3. Comparison of execution times of the sequential and parallel versions for one timestep.

3. Experiments and Results

All our experiments are carried out on a 2.6 GHz AMD© Phenom$^{\text{TM}}$ Quad-Core Processor with 2 GB of RAM per core running Open SUSE Linux, and an NVIDIA GTX280 GPU with 30 multiprocessors, 240 cores and 1 GB of RAM, using the version 3.1 of the CUDA driver compiler.

To study the scalability, we present the performance results for different data sizes. We collect simulation results for clusters with a Plummer density profile and sizes ranging from 10^6 to 7×10^6 stars, encompassing nearly all globular cluster sizes. Figure 3 compares the GPU and CPU run-times. We see that we obtain an average speedup of 28X and very good scalability. A quick calculation from the run-times and the data sizes in Figure 3 shows that the GPU scalability follows the kernel's complexity $O(NlogN)$.

References

Collins, J. C. 2008, Army Research Laboratory, 4, 41

Hammersley, J. M., & Handscomb, D. C. 1964, Monte Carlo methods (Methuen, London)

Heggie, D., & Hut, P. 2003, The gravitational million-body problem (Cambridge University Press)

Hénon, M. H. 1971, Astrophysics and Space Science, 14, 151

Hurley, J. R., Tout, C. A., & Pols, O. R. 2002, MNRAS, 329, 897. `arXiv:astro-ph/0201220`

Joshi, K. J., Rasio, F. A., & Zwart, S. P. 2000, The Astrophysical Journal, 540, 969