

Optimizing Monte Carlo Simulations of Star Clusters with CUDA

Honors Thesis
Matthew Bierbaum
Advisor: Fred Rasio

Northwestern University Dept. of Physics & Astronomy
May 11th, 2009

1. Introduction

1.1 Cluster Dynamics

The evolution and dynamics of star clusters is fundamental to many astrophysical problems that remain unresolved today. Clusters are thought to be the birthplace of all stars of which we know. Through dissipation effects early in their evolution, they are thought to even form the isolated stars that we can see today. The very high stellar densities inside of these star clusters cause them to contain a relatively high number of primordial binary stars and hence make them the birthplaces and homes of many interesting phenomena and systems (Hut et al. 1992). Many of these systems have been observed experimentally but have not been fully explained theoretically.

These phenomena are quite varied and provide the opportunity of insight into the interesting physics that occur in them. One such system is that of blue stragglers. These are stars that are brighter and bluer than other stars in the same population and are often associated with globular clusters. The currently favored formation theory is that of a collision between two lower mass stars to form a evolutionally-delayed more massive star. Information about collision rates and efficiency of formation of these systems could lead to a better understanding of the collisional process (Sills et al. 2009). The collisional conditions found in clusters are also thought to lead to the formation of intermediate mass black holes (IMBH). If, in less than 3 Myrs after star birth, a very massive star ($M_{\text{star}} \sim 400 - 4000 M_{\text{sun}}$) is formed through many smaller collisions then it could potentially collapse to form an IMBH (Freitag et al. 2006). These IMBHs could ultimately provide a good source for gravitational waves if found in binaries (Fregeau et al. 2006).

The binaries found in clusters are also thought to lead to the intense X-ray sources that have been found in the cores of some star clusters (Elsner et al. 2008). Thought to come from X-ray binaries, very short period pulsars such as millisecond pulsars have been associated with some star clusters as well. Measurements from these pulsars can help determine interstellar medium properties as well as distinguishing cluster core mass and properties of pulsars in general (Camilo et al. 2000).

Along with providing insight into the physics of these systems as well as general gravitational N-body interactions, clusters can reveal information about the early history and current cosmology of the Universe. For example, globular clusters contain some of the oldest stars that we can observe. Using observations of clusters from our own Milky Way, Kraus et al. 2003 were able to set a lower limit on the age of the Universe to 11.2 Gyr. Also, the determination of interstellar medium properties from pulsars could help place constraints on cosmological parameters.

1.2 N-Body Simulations

In order to study the dynamics of clusters, one can perform gravitational N-body simulations with varying initial conditions and many layers of additional physics such as stellar evolution and collisions. However, the distances and timescales involved in these simulations make them very difficult to perform computationally in a reasonable amount of time. The entire scaling of the cluster can be on the order of 10 pc (10^{20} cm) while interactions between binary stars that cause mass loss from the cluster through ejection occur on the order of 10^{2-4} cm. In the same vein, if we try to include stellar evolution, we find a wide disparity in scaling of time. Core collapse can be held off for longer than 10^{10} yr through processes such as binary burning, while a single binary interaction can take as little as 10^0 yr. On top of this, the number of stars that should be used to simulate a reasonably sized cluster needs to be on the order of $10^5 - 10^7$, which is itself computationally demanding for simple direct N-body simulations. Many methods have been used to try to cope with these extremes. These include direct N-body which calculates inter-body forces and integrates Newton's equations of motion, direct integration of the Fokker-Planck equation, and the method our group has chosen, the orbit-average Monte Carlo method.

Though all of these methods ultimately track the movement of stars in the cluster over its lifetime, the details that go into each of them are quite different. In the direct N-Body method, the total gravitational force is found on each star and then integrated to find the total motion of the star in one time-step. These equations follow Newton's equations of motion

$$\vec{F}_j = \sum_{i \neq j}^N \frac{Gm_i m_j}{|\vec{r}_{ij}|^2}; \quad \ddot{\vec{x}}_j = \frac{\vec{F}_j}{m_j}; \quad (1-2)$$

which can be solved using a variety of numerical integration techniques, some as simple as Euler's method. However, in order to best preserve energy in the system, the integration time-step must be small in comparison to the total orbital period, particularly when there are many close interactions. This means that the total force must be summed many times for each orbit. By inspection we see that this calculation alone grows with N^2 , putting a limit on the number of stars that can be simulated in a reasonable amount of time as well as the amount of time a simulation may run in total. By making assumptions about clusters, one is able to simplify the system of equations by beginning with the Boltzmann equation and working towards the Fokker-Planck (FP) equation. The discussion here about the FP and Monte Carlo methods follow Freitag 2008.

The basis of the FP method is the collisionless Boltzmann equation which describes the continuity of the phase space distribution of an ensemble of particles. This phase space is usually denoted $f(\mathbf{x}, \mathbf{v}, t)$ and can be thought of a probability distribution function for finding a particle in a certain point of the position and velocity phase space at a time t . However, clusters are far from collisionless and so this equation must be modified to account for close interactions that create the effect of relaxation in the cluster.

When we do this and assume that these interactions are short-term and small with respect to the overall motion of the particles, we arrive at the Fokker-Planck equation for phase space distributions in $f(\mathbf{x}, \mathbf{v}, t)$. We could stop here and evolve a collection of stars by directly integrating the FP in phase space as a function of time (Chang et al. 1970). However, this turns out to be very difficult, and the system can be simplified yet again when we make the assumption that a cluster is spherically symmetric in all of its properties and that it evolves only slightly over a dynamical time. Now, we can write the phase space as a function of the star's specific orbital energy and angular momentum, $f(\mathbf{x}, \mathbf{v}) = F(E(\mathbf{x}, \mathbf{v}), J(\mathbf{x}, \mathbf{v}))$ as these two parameters are able to fully describe a particle's position and velocity via the two equations

$$E = \phi(\vec{x}) + \frac{1}{2}(\vec{v}^2); \quad J = (\vec{x}) (\vec{v}_t) \quad (3-4)$$

where \mathbf{x} is the usual position vector, \mathbf{v} is the velocity, and v_t is the tangential component of the velocity. The function ϕ is the smoothed gravitational potential of the entire cluster. This is called the orbit-averaged method and is the one adopted by our group.

1.3 Monte Carlo Method

This method uses the assumptions of the orbit-average Fokker-Planck equation but does not involve direct integration. Instead, the FP equation is solved by applying the Monte Carlo method to a group of particles. This was initially done by Henon (1971a,b) and has, since his initial work, been greatly improved to include the extremes of timescales and sizes mentioned previously. In a simple case, the method begins by setting up initial conditions based on an isotropic distribution for mass and position of each star. The energies and angular momentums are computed and then perturbed slightly based on an analytical model for close gravitational interactions that occur in a star's neighborhood, providing the small changes that lead to relaxation over many orbits. Once the new values of E and J are found, the star is moved to a position and velocity, which are consistent with both these values. For example, based on equations (3-4), the new magnitude of radial velocity should be chosen such that:

$$Q \equiv v_{rad}^2 = 2E - 2\phi(R) - \frac{J^2}{R^2} \quad (5)$$

Then, its new energy and angular momentum are computed and perturbed once again in a loop that is the basis of the simulation. The timescale between each perturbation is that of the central relaxation time so as to capture both the dynamics at the edges of the cluster as well as the core (Joshi 2000).

The very interesting and useful aspect of the Monte Carlo method is its ability to incorporate extra physics easily into the simulation (Freitag et al. 2001). The code in our

group, for example, tracks the evolution of single-single, binary-single, and binary-binary stars using direct N-body in between time-steps of the main MC dynamics. This allows us to follow even the smallest timescales in the cluster while also running simulations for several gigayears. Also, we have been able to include stellar evolution for each of these systems as well as single stars in order to track the evolution of collisional products.

While the code is able to evolve clusters relatively quickly compared to standard methods such as the direct N-body, it can still take on the order of days to weeks to run a full simulation, depending on the number of stars in the cluster and the physics that are pertinent to the run. Therefore, we turn to alternative architectures to try to improve the performance of CMC so that we can arrive at results faster and be able to simulate larger N values for our clusters. In particular, we port the most computationally demanding sections of our code over to the computer's graphics processing unit, or GPU.

2. The GPU

In recent years, the ordinary GPU that is a part of most modern computers has become a very powerful computational tool. These processors are designed to render 3D scenes for computer games as quickly as possible. As such, their architecture allows many small operations to be done on a set of data very quickly by employing massively data parallel computation. This, along with their relatively high abundance due to the gaming market, makes GPUs very attractive for use in high-performance computers (HPCs) as a way to cheaply improve the performance of codes without building an extremely large CPU cluster (Owens et al. 2008). For example, our group was able to expand the compute capability of our existing cluster, "fugu", by over 30 TFLOPS by adding only 10 GPU nodes compared to the existing 3.4 TFLOPS spread across 79 nodes ("The Fugu Cluster" 2009). The high density of computational power also decreases physical storage and cooling costs as well as the cost to power the cluster. In many ways, the GPU can be seen as a more cost efficient computational tool (Owens 2007).

2.1 Hardware

The GPU hardware is very similar to any standard CPU hardware in that the basic chip is organized into arithmetic logic unit's (ALUs), control units, and a cache. Our description of these items will follow NVIDIA's CUDA Programming Manual, 2008. Connected to the GPU is a larger memory storage area called the DRAM or dynamic random access memory where data such as the program you are running are stored. There are major differences, however. First, the GPU has much more die area dedicated to ALUs. In the case of the GTX 280, there are in fact 240 ALUs that are grouped into 30 multiprocessors (MP) that each consist of eight scalar processor (SP) cores. Each of these multiprocessors has the basic parts of the CPU - ALUs, a control unit, and a cache. The cache for a multiprocessor is called shared memory and is on the order of 16KB in size. This is only available to the ALUs in a single multiprocessor. Along with this, there is a larger constant memory and texture memory, on the order of 64KB and 8KB respectively, which are available to any of the multiprocessors and so allows for cross-talk between the ALUs. Finally, the individual ALUs also contain a group of 8192

registers that they can use. This is compared to a modern four-core system that has four ALUs each with a set of registers and usually three levels of cache - L1, L2, and L3 which are on the order of 512KB to 3MB.

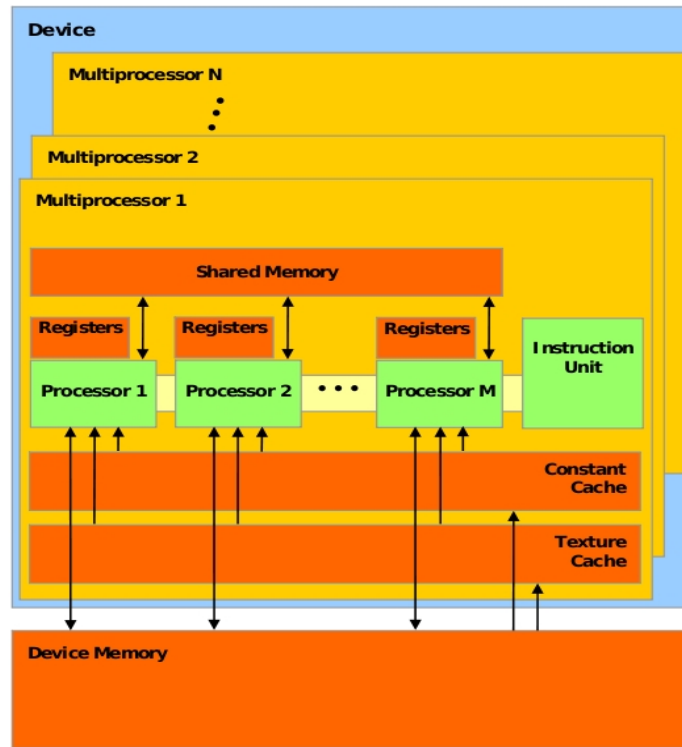


Figure 1: Schematic for the architecture of the GPU. NVIDIA CUDA Programming Guide V2.0, 2008.

Off of the system chip, there is also a DRAM on the video card. This memory is about 400x slower in access and read times than the onchip memory. However, it is able to hold much more information, on the order of 512MB to 1.5GB making it the main storage area on the GPU. One of the goals in the optimization of a CUDA program is to hide this high memory latency from copying global memory to the ALU by making use of the constant and shared memory for temporary storage. Another consideration about this memory is that it is physically distinct from the DRAM that the CPU accesses normally. The programmer must copy all the required data for a computation to the GPU's DRAM which involves moving data across the PCI-x16 lanes. The theoretical throughput of these buses is around 4.3GB/s while, in actuality, the GPU is able to accommodate around 2.4GB/s.

In order to maximally use all of the ALUs on a GPU, there is a complex system of schedulers that allow for 65,535 concurrent threads to run across all 240 of the processors of the GPU. To better organize this structure, there are a number of levels of subdivision. The kernel itself is placed in a single grid that is divided into blocks of threads. At most, there can be 512 threads to a block and $65535 / 512 = 128$ blocks per grid. These blocks don't have a very physical meaning besides a means of organization of threads for the scheduler. It is not made such that one block exists on one multiprocessor or a similar

organization. In fact, there can be as many as 8 active blocks exchanging runtime on the same multiprocessor or as many as 768 threads active on the same MP. There are many factors which might influence the dimensions of the blocks and grid that is chosen. For example, if your kernel is very large and needs to use 112 registers per thread, then at maximum, there can be 585 threads active on a given MP. If a grid size of 16 blocks is then chosen with 512 threads per block, only one block will be active. In fact, if the number of registers is not regarded, the kernel can fail to launch due to lack of resources.

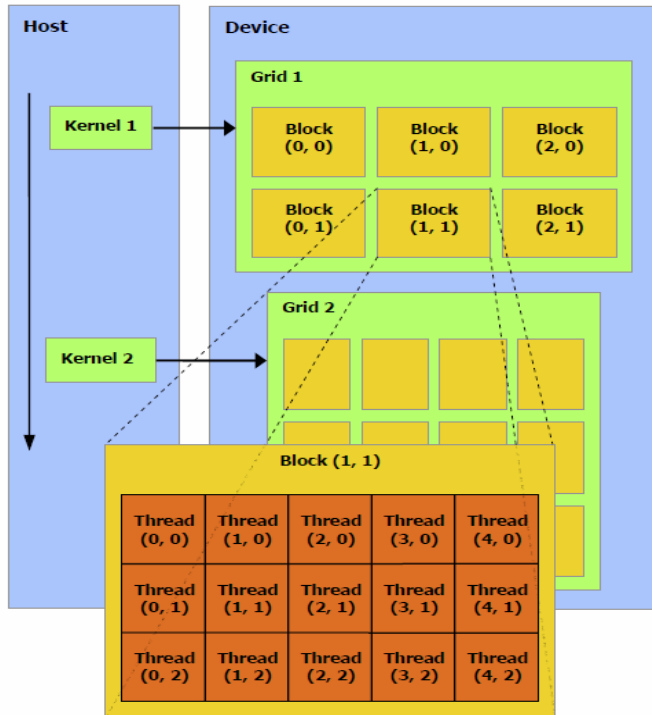


Figure 2: Layout of the thread hierarchy CUDA-enabled devices. The left displays how the host CPU may call several kernels asynchronously, each running on a different grid. NVIDIA CUDA Programming Guide, V2.0.

2.2 CUDA

In order to promote use of the GPU as a generally programmable graphics processing unit (GPGPU), NVIDIA developed the Compute Unified Device Architecture (CUDA) as an extension to the C programming language. This allows programmers to more easily control aspects of their program on the GPU to more fully use the hardware to its ability. Previously, programmers needed to use assembly language and manipulate shader functions to achieve the same effect (Harris 2005). CUDA comes with a modified g++ compiler called *nvcc* which is able to compile both C and C++ code. Along with this, it comes with the CUDA libraries which help initialize the devices, debug, profile, and time the code. Ultimately, all of these tools allow data to be copied to the GPU and a kernel to be called asynchronously on the GPU which in turn performs some operation on the data. The results can then be returned to the host CPU through more memory copy functions which bring the memory back from the GPU to CPU (NVIDIA 2008).

3. CMC-GPU

The cluster monte carlo - gpu code was written by optimizing only one function in the entire program. This is *FindZero_Q* which finds the zeros of equation (5) via root bisection in order to decide the velocity and position of each star in the next time-step. For this function, the values for energy, angular momentum, and the potential are all computed on a grid based on the stars position, creating a negative quadratic. These values only depend on the masses, radial positions, potentials, energies, and angular momentums of itself and its neighboring stars. Therefore, in order to parallelize this, we chose to assign one thread to perform the root bisection for one star each, splitting the calculation into grids of size 256 threads/blocks * 28 blocks/grid = 7168 threads / grid that can run concurrently. At kernel launch time, many of these grids are launched and are run asynchronously until everyone of them has finished.

In order to avoid warp divergence by using “if” statements, the root bisection was compressed into an algorithm that requires no path choosing. Instead, it shifts the bounds of the search by using simple arithmetic along with the fact that the potential is well behaved and resembles a negative quadratic. The basic code listing as a comparison between the functions is found in Listing 1 and 2, simplified for readability. Notice that in fact, the code could be written out entirely iteratively, independent of the star index, how well the search is doing, and how many stars are in the cluster.

```

long FindZero_Q(long j, long kmin,
               long kmax, double E, double J){
    long ktry;

    if(FUNC(j, kmin, E, J)<FUNC(j, kmax, E, J)){
        do {
            ktry = (kmin+kmax+1)/2;
            if (FUNC(j, ktry, E,J)<0){
                kmin = ktry;
            } else {
                kmax = ktry-1;
            }
        } while (kmax!=kmin);
    } else {
        do {
            ktry = (kmin+kmax+1)/2;
            if (FUNC(j, ktry, E,J)>0){
                kmin = ktry;
            } else {
                kmax = ktry-1;
            }
        } while (kmax!=kmin);
    }
    return kmin;
}

```

Listing 1: The original root bisection code. The basis of the search is divergence which degrades GPU performance.

```

__global__ void cuFindZero_Q(long start, long nstar, double *cu_m,
double *cu_r, double *cu_phi, double *cu_E, double *cu_J, long
*cu_kmin, long *cu_kmax, long *cu_ktemp)
{
    long si = threadIdx.x + blockDim.x * blockIdx.x + start;
    long j = si;

    double E, J, t;
    long ktry, kmin, kmax, sa, sb, k1, k2;

    E = cu_E[j];
    J = cu_J[j];
    kmin = 0;
    kmax = si;

    t = FUNC(j, kmin, E, J, n, cu_m, cu_r, cu_phi);
    sa = sign(t);
    t = FUNC(j, kmax, E, J, n, cu_m, cu_r, cu_phi);
    sb = sign(t);

    for (long i=0; i<=MAX_LOOPS; i++)
    {
        ktry = (kmin+kmax+1)/2;
        t = FUNC(j, ktry, E, J, n, cu_m, cu_r, cu_phi);
        kmin = (1 + sa*sign(t))/2 * (ktry - kmin) + kmin;
        kmax = (1 + sb*sign(t))/2 * (ktry - kmax-1)+kmax;
    }

    k1 = kmax;

    kmin = si;
    kmax = nstar;

    t=cuFUNC(j, kmin, E, J, n, cu_m, cu_r, cu_phi);
    sa = sign(t);
    t=cuFUNC(j, kmax, E, J, n, cu_m, cu_r, cu_phi);
    sb = sign(t);

    for (long i=0; i<=MAX_LOOPS; i++)
    {
        ktry = (kmin+kmax+1)/2;
        t=cuFUNC(j, ktry, E, J, n, cu_m, cu_r, cu_phi);
        kmin = (1 + sa*sign(t))/2 * (ktry - kmin) + kmin;
        kmax = (1 + sb*sign(t))/2 * (ktry - kmax-1)+kmax;
    }
    k2 = kmax;
    cu_kmin[j] = k1;
    cu_kmax[j] = k2;
}

```

Listing 2: The GPU implementation of *FindZero_Q*. No divergence is present though every reference to *cu_** is a global memory read. Both writes to global memory are made at the end of the function to ensure coalesced memory access.

4. Results

When profiled, *FindZero_Q* was placed at the top of CMC's call list at 22.7% of the total runtime of the program. Without turning on extra physics such as stellar evolution, binaries, or collisions there was an average speedup of 22.2% of the CUDA version as compared to the CPU implementation. This was achieved by finding the most optimized pair of grid dimension, block dimension pairs for the launch parameters of the kernel.

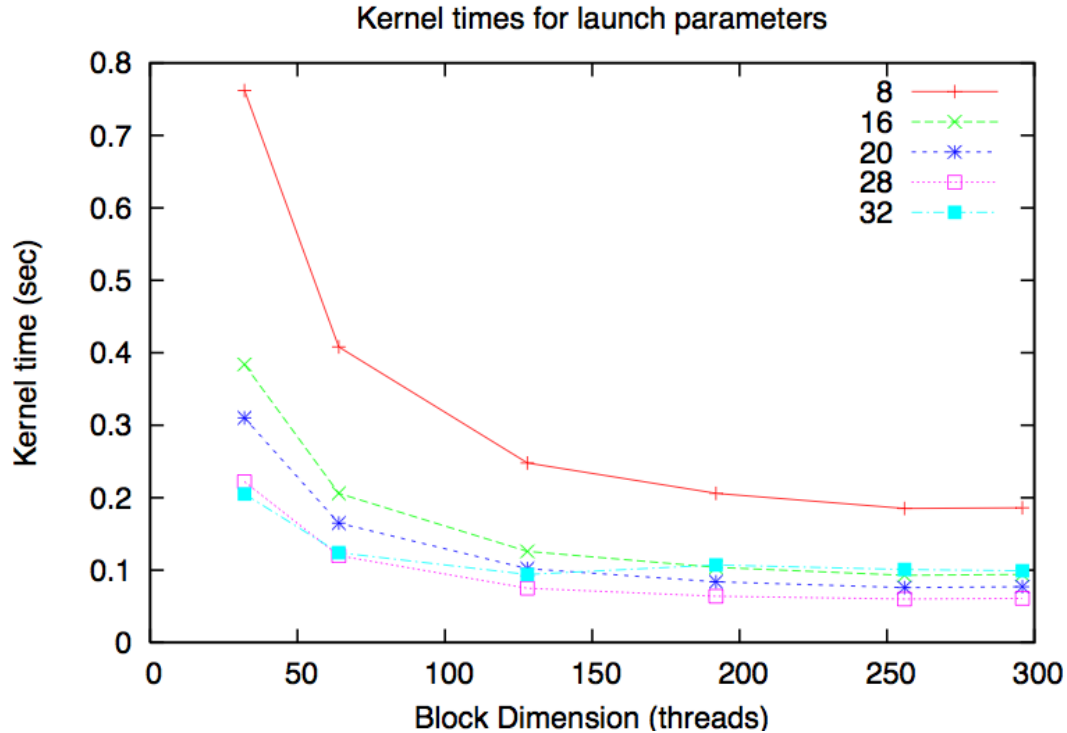


Figure 3: Total kernel runtime for $5 \cdot 10^5$ stars. The lines are various grid sizes, the optimal being 28 blocks per grid. This is compared to a CPU runtime of 2.36 sec.

The slowest pair tested was (8, 16) which severely under-loaded the GPU. The limiting number of blocks a MP can run at concurrently is 8 and so the number of total threads running on the GPU was far below the limit imposed by the number of registers. The best configuration was that of (28, 256) which loads the GPU the best without causing launch failure due to too few resources. On the other end, (32, 256) was significantly slower than optimal because threads were left waiting for a MP on which to run. With the optimal configuration, however, the runtime for all stars to undergo *FindZero_Q* dropped from 2.36 sec to 0.059 sec, a forty fold improvement. This strongly agrees with the speedup shown in the total runtime – $22.7 / (22.7 - 22.2) = 45.4x$ faster, assuming all of the improvement came from the single function that was optimized.

Another set of performance numbers that is helpful to look at is the number of floating-point operations performed per second, or FLOPS. To calculate this, we have to first find the total number of operations that our kernel performs in one run. For one star, there are 2916 FLOPs making a total of 1.458GFLOPs for a cluster of $5 \cdot 10^5$ stars. For the fastest kernel configuration, this translates into a performance of 24.7 GFLOPS. It turns out that this is very close to the memory access bandwidth between the GPU and its onboard memory of 20GB/s. This makes sense because our calculations are severely memory-bound in that there is almost a one to one correspondence between FLOPs and reads from global memory.

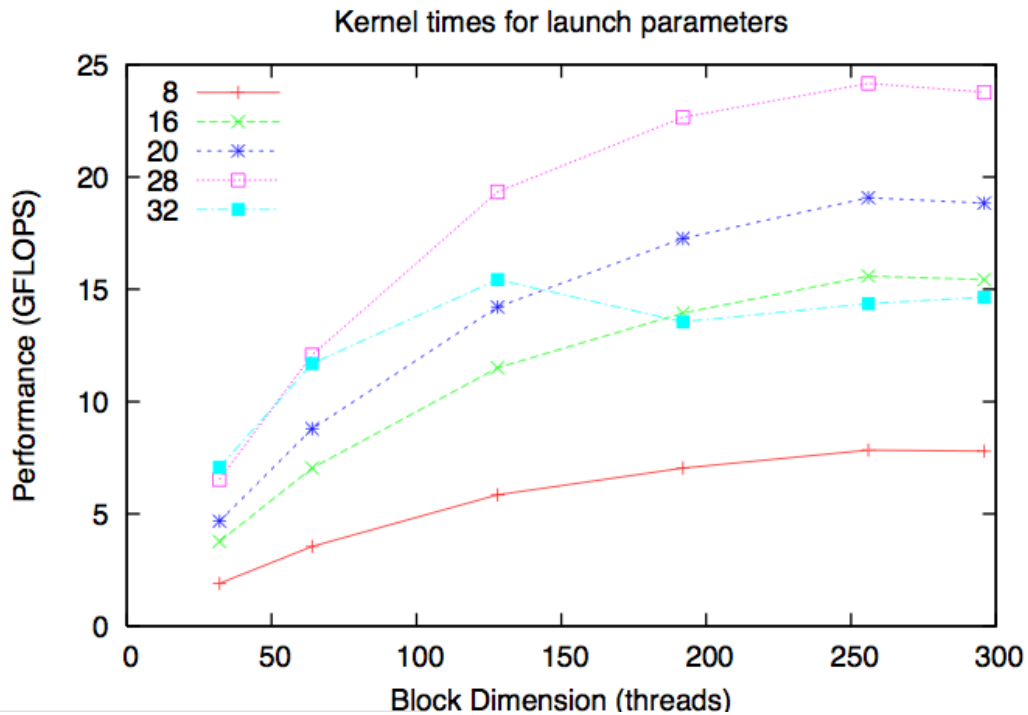


Figure 4: Kernel performance shown as a function of grid and block dimension measured in GFLOPS. Peak performance is around 24.7 GFLOPS, right around the same speed as the memory bandwidth of the GPU.

4.1 Discussion

This original approach to optimizing *FindZero_Q* turns out to provide adequate performance so that further optimizations in this area would be largely unnoticeable. Even though the implementation uses no shared memory and is executed entirely in double precision, there is very little room for improvement. The tradeoff between time spent on the function and the time gained in improvements would be minimal.

If, however, the function took a much larger percentage of the total runtime, there are a few ways that the CUDA function could be improved to get even more performance out of the GPU. These would mainly focus on minimizing the use of the much slower global memory and occupying more of the GPU by using floating-point numbers. In order to make better use of the shared memory, we could group the threads into sections of stars that would follow the same bisection pathway. For example, if we had 16 stars, we could load the information at grid points 0, 4, 8, 12, 16 for threads that would start at 0, 8, or 16. Then the information for 2 to 3 steps in any of those bisections would be preloaded into shared memory and access would be much lower for those bisections. However, if we again look at the real bisection in *FindZero_Q* with 1 million elements, we find that at most, 6 out of 24 steps can be loaded at once into shared memory before too much divergence in the bisection path makes preloading inefficient. 8192 bytes are available for each multiprocessor and 5 pieces of data are needed at each grid point leaving $8192 \text{ bytes} / (5 \text{ doubles} * 3 \text{ bytes} / \text{double}) = 546$ grid points. This only allows for 6 levels of search which at most represents an increase of 25% performance assuming memory read time drops to zero for those steps.

Another possible method to increase performance is to use double arithmetic only when absolutely necessary. The ratio of single precision to double precision capable processors is 8:1. Theoretically, using entirely single precision floating-point operations could speed up the kernel by 87%. There are limitations here as well including that SP division is not IEEE standard on the GPU and has a ULP error of 2 places whereas DP division is compliant with full IEEE standards. In the kernel, 7 out of 16 DP operations are in fact division. While all of the other DP operations such multiplication and addition could potentially be implemented using two SP floats, division will always contain too much error when emulated with two SP numbers. Therefore, at the most optimistic level, we can expect an improvement of about 50% because we can at most remove about half of the DP operations. However, this is very optimistic because the emulation of DP floating point numbers introduces many other operations (on the order of 30) as well as twice the number of global memory reads to retrieve two numbers instead of one. A much more conservative estimate of total improvement is around half the optimistic level, or 25%, due to the double memory reads.

Therefore, the naïve approach to improving performance of CMC on the GPU worked quite well. Further improvements would not provide very much performance gain for the time required to implement the procedure. We have seen a 40x speed improvement in the *FindZero_Q* function resulting in an overall speedup of 22.2% for the entire runtime.

These improvements will have a number of implications for our group. In the search for collisional regimes where the formation of IMBHs is possible, we will be able to expand our parameter space to include larger values of N to better cover all possible clusters including the very massive 47 Tuc. These runs with large N typically take on the order of a few weeks and so the GPU will be able to reduce the total runtime by a few days. Also, with this gain in speed, we will be able to decrease the integration time-step of the code in order to simulate the dynamics of globular clusters with IMBHs in the core. These massive objects make relaxation timescales much smaller and so more dynamical time-steps must be taken to resolve orbits properly.

Other ongoing projects in the group that might benefit from these code improvements include the verification of blue straggler formation by the collisional process mentioned in Sills et al. 2009. These simulations require many additional layers of physics including stellar evolution and collisions in order to fully capture the physics in question. The speed improvements would help to increase the number of runs possible to search the parameter space of blue straggler creation. In addition, there are more general dynamical star population studies happening in the group. To do so, populations of various compact objects such as black holes and neutron stars as well as their binary forms are tracked dynamically through a cluster's evolution. Dynamical friction sends these objects to the core of the cluster very early, causing the simulation to slow down greatly due to the decrease in central relaxation time. These simulations can therefore last for over a month. A 22% speedup can take off a week of computational time and simulations can then be run to completion in a more reasonable time frame.

This CMC-GPU is one of the first major GPU codes written in the group that takes advantage of the new GPU nodes that have been added to fugu. Soon, it will be employed by members of the group on these various projects. Hopefully this will lead to a wider adoption in the group of the GPU as a computational tool.

Acknowledgements

I would like thank several people who made this research possible. First, thank you to my family and the Integrated Science community for their eternal support and encouragement. I'd like to thank the entire Theoretical Astrophysics group, especially Professor Fred Rasio, for their many hours of support and guidance throughout this project. Much of this research was supported by a National Science Foundation MRI award to Professor Kalogera.

- Camilo, F., Lorimer, D.R., Freire, P., Lyne, A.G., Manchester, R.N. *Observations of 20 Millisecond Pulsars in 47 Tucanae at 20 Centimeters*. ApJ 535, 975-990 (2000)
- Chang, J.S., Cooper, G. *A Practical Difference Scheme for Fokker-Planck Equations*. Computational Physics 6, 1 (1970)
- Elsner, R.F., Heinke, C.O., Cohn, H.N., Lugger, P.M., Maxwell, J.E., Stairs, I.H., Ransom, S.M., Hessels, J.W., Becker, W., Huang, R.H., Edmonds, P.D., Grindlay, J.E., Bogdanov, S., Ghosh, K., Weisskopf, M.C., *Chandra X-Ray Observatory Observations of the Globular Cluster M71*. ApJ 687, 1019-1034 (2008)
- Fregeau, J.M., Larson, S.L., Miller, C., O'Sharughnessy, R., Rasio, F.A., *Observing IMBH-IMBH Binary Coalescences via Gravitational Radiation* ApJ 646, 135-138 (2006)
- Freitag, Marc. *Fokker-Planck Treatment of Collisional Stellar Dynamics* Lect. Notes Phys. 760, 97-121 (2008)
- Freitag, Marc. *Monte-Carlo Models of Collisional Stellar Systems* Lect. Notes Phys. 760, 123-158 (2008)
- Freitag, M., Benz, W. *A new Monte Carlo code for star cluster simulations: I. Relaxation* Astronomy & Astrophysics 375, 711-738 (2001)
- Freitag, M. Gurkan, M.A., Rasio, F.A. *Runaway collisions in young star cluster – II. Numerical Results*. Royal Astronomical Society 368, 141-161 (2006)
- Harris, Mark. *GPGPU: General-Purpose Computation on GPUs*. Game Developers Conference (2005)
- Hénon, M., *Monte Carlo Models of Star Clusters* Astrophys. Space Sci. 13, 284 (1971)
- Hut, P., McMillan, S., Goodman, J., Mateo, M., Phinney, E.S., Pryor, C., Richer, H.B., Verbunt, F., Weinberg, M. *Binaries in Globular Clusters*. Astronomical Society of the Pacific 104, 981-1034 (1992)
- Joshi, K.J., Rasio, F. A., Portegies Zwart, S. *Monte-Carlo Simulations of Globular Cluster Evolution. I. Method and Test Calculations*. ApJ 540, 969-982 (2000)
- Krauss, L.M., Chaboyer, B. *Age Estimates of Globular Clusters in the Milky Way: Constraints on Cosmology*. Science, 299, 65-70 (2003)
- NVIDIA Compute Unified Device Architecture Programming Guide Version 2.0*. NVIDIA Corporation (2008)

Owens, J., *GPU Architecture Overview*. SIGGRAPH 2007

Owens, J.D., Houston, M., Luebke D., Green, S., Stone, J.E., Phillips, J.C., *Graphics Processing Units – Powerful, programmable, and highly parallel – are increasingly targeting general-purpose computing applications*. IEEE Xplore 96, 5 (2008)

Sills, A., Karakas, A., Lattanzio, J. *Blue Stragglers After the Main Sequence*. ApJ 692, 1411-1420 (2009)

Theoretical Astrophysics Group. “The Fugu Cluster” 2009 Accessed: 5 May 2009. <<http://www.astro.northwestern.edu/Theory/Fugu/fugu.php>>