Parallel N-Body Simulation using GPUs

Francisco Chinchilla fchinchi@cs.unc.edu
Todd Gamblin tgamblin@cs.unc.edu
Morten Sommervoll morten@cs.unc.edu
Jan F. Prins prins@cs.unc.edu

Department of Computer Science
University of North Carolina at Chapel Hill
http://gamma.cs.unc.edu/GPGP

Technical Report TR04-032
December, 2004

Abstract

We present a novel parallel implementation of N-body gravitational simulation. Our algorithm uses graphics hardware to accelerate local computation, and is optimized to account for low bandwidth between the CPU and the graphics card, as well as low bandwidth across the network. The number of bodies that can be simulated with our implementation is limited only by the memory of the graphics card, and results for small clusters indicate that it will scale well across larger numbers of nodes. Finally, we show that our algorithm significantly outperforms a comparable CPU implementation. Heretofore, commodity graphics hardware has been used mainly for graphics and visualization applications. This work shows that it can also be used effectively for scientific computation.

1 Introduction

Programmable Graphics Processing Units (GPUs) are becoming ubiquitous on consumer PCs. Successive models have increasingly rich feature sets, enabling the GPU to be used effectively as a coprocessor for general purpose computation. The GPU is well suited to this task, as it provides much higher potential floating point performance at a lower cost than today's CPUs. Also, the performance of GPUs is increasing at a rate faster than Moore's law, so harnessing their power should prove to be a good investment. Furthermore, commodity hardware is being used more and more to construct high performance clusters of machines.

The intersection of these trends makes the GPU an appealing option for accelerating scientific computation on cluster systems. However, while the performance of the GPU is increasing quite rapidly, memory performance is increasing at a slower rate. Using the GPU as a coprocessor necessitates read-back of data to the CPU. Current graphics processors are not optimized for this operation: it causes a stall in the GPU. Programmers must be careful to use it sparingly.

In this paper, we investigate the potential of the GPU to speed up N-body gravitational simulation. N-body is an important problem in cosmology and astronomy, as it enables scientists to visualize and understand the behavior of galaxies, nascent planetary systems, and the evolution of the universe (to name a few applications). The problem requires significant computational power, as each body may, in the worst case, have a strong effect on every other body in the system, leading to $O(n^2)$ performance. Large simulations can thus take many CPU hours to complete. Accelerating this process will speed the pace of discovery.

1.1 Related Work

Much work has been done on using the GPU to accelerate scientific computation [9, 6, 5, 12, 10, 11]. Each of these papers covers a specific application and its implementation on the GPU. None of these have yet covered N-body simulation, and none have covered GPU applications on clusters.

A group at Stony Brook University [1] has constructed a cluster from 32 Dual 3Ghz Pentium Xeon systems equipped with nVidia GeForce 5800 graphics cards. They have implemented the Lattice Boltzmann method to run on the cluster's GPUs. They report a speedup of 4.6 times the speed of a CPU implementation, and this is approximately half the performance of the algorithm on a 32 node IBM Power4 BladeServer cluster. Also, they find that a single GPU is 6.6 times faster than a single CPU in the system. Had they used a GeForce 6800, this number would jump to 16.6.

Work has been done on analyzing the memory performance of graphics hardware. Igehy, *et al.* [4] present a graphics architecture optimized for rendering. While the system they present is not implemented directly in any commercial system, the optimizations they make and the tradeoffs they analyze are instructive in understanding design decisions and expected memory reference patterns in modern graphics hardware. Fatahalian, *et al.* [2] look at matrix-matrix multiplication on modern GPUs, and discuss the bandwidth limitations of the hardware. These two papers are instructive in understanding optimization on the GPU.

Finally, a tremendous amount of work has gone into optimizing the performance of N-body algorithms for traditional parallel architectures [8, 3, 13]. These algorithms use far more sophisticated numerical methods and optimizations than does our implementation. As we are interested only in showing that N-body can be run with a good speedup across multiple GPUs in a cluster, we chose a simpler algorithm for this first attempt.

1.2 Main Contribution

The main contributions of this paper are outlined as follows:

Algorithm: We present an algorithm for all-pairs N-body simulation and show how it can be adapted and optimized for the GPU. We describe the steps we took to optimize the application both for low CPU-GPU bandwidth and for low bandwidth over the network.

Cluster: We describe how to build an ad-hoc cluster out of commodity hardware and graphics cards.

Speedup: We show that by using a cluster of GPUs, the speed of an N-body simulation can be increased by al-

most eight times. We also show that with our algorithm, this speedup scales linearly with the number of nodes in the cluster.

1.3 Organization

The rest of this paper is organized as follows: In §2 we outline the N-body problem. We present an N-body algorithm for a single GPU in §3. In §4, we describe the implementation of our GPU cluster. In §5, we present our algorithm adapted for use on multiple GPUs. We improve on this algorithm in §6 and optimizes it to limit CPU-GPU data transfer. We discuss our results in §7, and address the issue of numerical error in N-body systems. Finally, §8 outlines our conclusions and future directions for this work.

2 The N-body Problem

In this section, we give a brief overview of the N-body gravitation problem.

The initial inputs to the problem are a set of n bodies, $b_1, b_2, ..., b_n$, where each body b_i has a mass m_i, a velocity v_i, and position r_i. The distance between any two bodies b_i and b_j is written r_{ij}, and the gravitational force on b_i as a result of b_j is written f_{ij}.

Let the total gravitational force on a body b_i be written f_i. For each iteration, given a timestep Δt, we want to compute the new positions of each body after has elapsed. This can be done in three phases:

1. First we compute partial forces f_{ij} for all pairs of bodies:

$$f_{ij} = \frac{G m_i m_j r_{ij}}{|{r_{ij}}^3|}, \quad i \neq j \tag{1}$$

   G here is the universal gravitational force constant, and it is equal to $6.673e-11\ m^3/kg\ s^2$.

2. Next, we compute the total force f_i on each b_i:

$$f_i = \sum_{j, j \neq i} f_{ij} \tag{2}$$

3. Finally, we update the velocity v_i and position r_i of each body using the classical force equation, $F = ma$:

$$\Delta v_i = \frac{f_i \Delta t}{m_i} \tag{3}$$

$$v_i' = v_i + \Delta v_i \tag{4}$$

$$r_i' = r_i + v_i \Delta t + \frac{\Delta v_i}{2} \Delta t^2 \tag{5}$$

Now, we have the updated positions in r_i, and can repeat for another timestep Δt. To measure the performance of an N-body algorithm, we typically refer to the interaction rate, or the number of interactions between bodies we calculate per unit time, defined as:

$$R(n, t_k) = \frac{n(n-1)}{t_k} \tag{6}$$

Where t_k is the average time per iteration. We will use this metric for our results in §7.

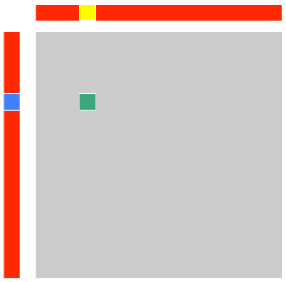


Figure 1: Force Matrix. Each of the red buffers is a texture containing bodies' positions and mass, and each pair (shown as yellow and blue) of bodies is interacted to find a partial force (shown in green).

3 Single GPU Implementation

In this section, we present a single-GPU N-body algorithm, built with code from [7]. This implementation follows the steps described in §2 very closely, and we focus mainly on the key issue of mapping this algorithm to the programming model of modern graphics hardware.

Our single-GPU implementation stores the bodies as a standard red, green, blue, alpha (RGBA) texture with either 16 or 32 bits per color value, depending on whether we use half or single precision floating point numbers for computation on the GPU. Each texel represents a single body. The R, G, and B channels are used to store the x, y, and z coordinates of the body's position, and the A channel is used to store its mass. This texture is stored on the GPU with arbitrary dimensions. Its total size need only be n texels.

For our force calculation, we render an nxn quad into a force texture, where each pixel rendered represents a partial force f_{ij}. We use a Cg fragment program and a lookup texture to compute the color values of each pixel in the force quad. The lookup texture is a standard RGBA texture of the same dimensions as the body texture. Its values map linear indices from $1..n$ to two-dimensional indices in the position texture of bodies to interact. For each rendered pixel (x, y) in the force quad, we look up the x^{th} and y^{th} texels in the lookup texture and use the resulting values to find the appropriate texels in the body texture. We then use the retrieved position and mass values to compute f_{ij} and store this value in the force texture.

Once all f_{ij} are computed, we compute each f_i using a parallel log reduction. We begin with the n x n force texture, and render a quad half its height into a texture. The i^{th} row in the rendered quad is the sum of the i^{th} and $2i^{th}$ rows in the force texture. We then successively render $log_2(n) - 1$ more quads in a similar fashion, where each is half the size of the previous one. When we are finished, we are left with an n x 1 quad, where the i^{th} element corresponds to an f_i.

Finally, we use very simple fragment programs to update velocities and positions. The velocity program takes as its inputs the f_i texture and the body texture, and it renders the updated v_i into the velocity texture. Similarly, the body program takes as its inputs the velocity and body textures, and renders updated body positions back into another body texture.

For simplicity, our simple single-GPU version is restricted to datasets no larger than 2048x2048. This is the maximum allowed texture size on a GeForce 6800 card. Our parallel algorithms, described in §5 and §6, demonstrate how this limitation can be circumvented.

Node(s)	CPU Configuration	GPU Configuration
0-2	3GHz Pentium 4 with Hyperthreading	nVidia GeForce 6800 GT
3	3.4 GHz Pentium 4 with Hyperthreading	nVidia GeForce 6800 Ultra
4	2.8 GHz AMD Athlon 64 FX-53	nVidia GeForce 6800 Ultra
5	Dual 2.8GHz Pentium Xeon	nVidia GeForce 6800
6	2.4GHz Pentium Xeon	nVidia GeForce 6800

Table 1: Configurations of Cluster Nodes

4 Cluster Description

In this section we describe the hardware configuration of our cluster. We also describe the software infrastructure used for message-passing between nodes.

4.1 CPU/GPU Configuration

Our cluster was constructed ad-hoc from computers around the department. We used all available machines with an nVidia GeForce 6800 series graphics card. We chose this card for four reasons:

1. Full 32-bit floating point support: At the time of the cluster's inception, the 6800 had the highest - precision floating point implementation of any commercially available card.

2. Memory: The 6800 series can be outfitted with up to 256MB DDR video memory. This was both the largest and highest-throughput memory available on any graphics card at the time of writing.

3. Speed: Save for the ATI Radeon X800 XT series, the GeForce 6800 series was the fastest GPU available to us at the time of writing. We chose the nVidia cards over the ATI cards primarily because the ATI cards support only up to 24-bit floating point numbers.

4. Programmability: The GeForce 6800 series offers support for custom vertex and pixel shaders, written in nVidia's Cg shader language. This enabled us to implement our custom N-body algorithm.

The names and configurations of all machines in our cluster are shown in Table 4. All nodes in our system ran Microsoft Windows XP Professional, with Service Pack 2. Although there are drivers for the GeForce 6800 series for both Windows and Linux, we chose to run Windows because of driver quality. In our experience, the nVidia drivers for Windows tend to stay slightly ahead of those for Linux in terms of performance optimizations.

4.2 Network Configuration

The network infrastructure for our cluster was Ethernet. For our measurements on the algorithm described in §5, we used an 8-port 3com Superstack-3 Gigabit Ethernet switch. For measurements on the algorithm presented in §6, we used a NetGear FS108 8-port 100baseT switch. During the experiments, the cluster machines had exclusive access to these switches so that there would be no interference from other traffic.

4.3 MPI Software

Communication between cluster nodes was accomplished using MPI (Message Passing Interface), the de-facto standard for inter-node communication in distributed-memory clusters. For our tests of the algorithm in §5, we used MPI/Pro from VerariSoft, Inc, a commercial implementation available for Windows, Linux, and Mac OS X. We used MPI/Pro for our first set of tests, but we experienced serious stability problems with the implementation. We then switched to MPICH a freely available, open-source implementation available from Argonne National Laboratory. MPICH proved to be remarkably robust, and we used it in tests of our final algorithm in §6.

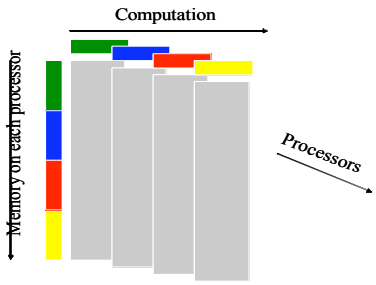


Figure 2: Initial Parallel N-body implementation.

5 Initial Parallel Implementation

In this section we discuss our first attempt at designing a parallel algorithm to run on the cluster. This algorithm was intended as a simple extension to the sequential algorithm described in §3. Our main goal was efficient scaling.

At the high level, our parallel implementation follows the same basic steps as the sequential algorithm. We first compute partial forces, then sum them, and then use this information to update positions and velocities. The key change is the way that work is divided up among processors in this version. For ease of illustration, we have again required certain restrictions on the input of this problem. We require that the total number of bodies n be of the form $n' * p$, where $n' = n/p$. For any one node in the cluster, we say that n' bodies belong to that node. Last, we require that n' be a power of two.

One simple way to think of our modification to the force computation is as a repeated application of the sequential case. Figure 2 provides an instructive illustration of this approach. We break up the all-pairs force texture into four chunks of size $n' * n$, each to be computed by a particular node. Each node can now use the sequential algorithm as a subroutine for computing chunks of n' bodies. To compute partial forces, we run the sequential algorithm p times, interacting our "own" n' bodies p times, once for each set of n' chunks belonging to a node.

The reduction we used in the simple GPU algorithm was fairly time consuming, as it required iterative rendering of quads. To minimize this overhead, rather than having each node compute a reduction on an entire column, we accumulate force values as we apply the sequential algorithm. Each pixel in the rendered quad is the sum of corresponding pixels

in chunks rendered so far. After p iterations, we are left with one chunk of pixels representing accumulated forces, and we perform the same reduction as before on this chunk.

Each node is left with the total forces on its own n' bodies. The node updates its own velocities and positions in the same way that was done in the sequential algorithm. The only difference here is that the operation is performed for the local n' and not the global n bodies. Once this update completes, we use `MPI_All_gather` to transfer all the positions to all processors. Once this is done, we are back to the start, and can begin another iteration.

This algorithm incurs additional overhead over the single-GPU implementation because it repeatedly swaps sets of n' bodies in and out of the GPU in the force accumulation stage. Note, however, that this is only for the multi-GPU case, as for a single GPU we only have to interact with ourselves, and we have all the information for our own bodies on-hand. Thus, there is no swap as we accumulate down columns for 1 GPU, but there is an additional copy overhead for multiple GPUs.

6 Optimized Parallel Algorithm

We made two main modifications to the algorithm in our final, optimized version. First, we removed the restrictions on input size that we had placed on previous implementations. This enabled us to put more bodies on the GPU, and to take much greater advantage of the GeForce 6800 series' 256 MB main memory. This change required a slight modification to the algorithm, as well. Our final algorithm works much like our unoptimized N-body algorithm, but instead of distributing each of the columns shown in Figure 2 to an individual GPU, it is capable of allocating multiple columns to the same GPU. This effectively removes the dependence on number of nodes, from which our earlier algorithms suffered, and it enables us to perform N-body computations with very high body counts on GPU-equipped machines.

With this first optimization, we also see some significant memory advantages. Earlier algorithms required that each buffer of bodies use a separate all-pairs render buffer, i.e. 1024 bodies on one GPU required a 1024x1024 all-pairs render texture where we accumulated the results. By using multiple local buffers of bodies on each node, we are able to reuse our all-pairs render texture.

The second optimization we performed with the new algorithm was changing the mechanism by which bodies not local to the GPU were transferred there. The algorithm described in §5 has one texture for bodies belonging to other nodes, and it swaps these in and out of the GPU during each iteration of force accumulation. We noted that on the GPU, doing two write-backs in different places was more than twice as slow as doing two write-backs to the GPU back to back. We changed our approach to store the positions of all other nodes on the GPU at all times, and we write all of them to the GPU in a batch, once per timestep.

7 Results

In this section we describe our results for all of our implementations of N-body for the GPU. We compare interaction rates, as well as speedup and efficiency on multiple GPUs. We also compare our interaction rates to those of an equivalent single-CPU algorithm, and to the interaction rates of a parallel CPU algorithm. Finally, we examine the numerical stability of our algorithm as compared to a CPU implementation.

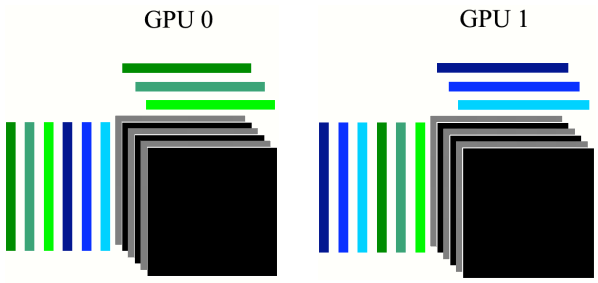


Figure 3: Optimized N-body algorithm. Black squares represent force matrices on different processors, and different color rectangles represent different subsets of body positions.

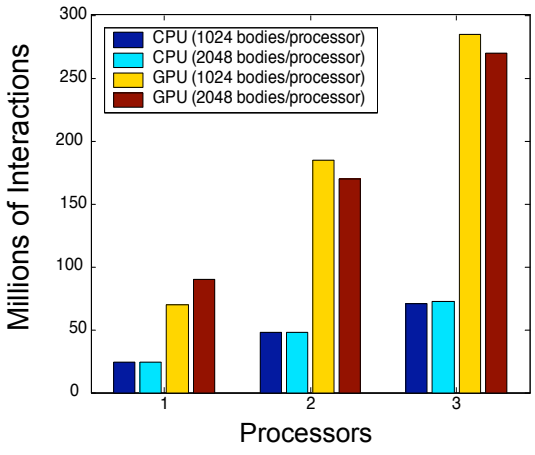


Figure 4: Single GPU vs. First Parallel Implementation

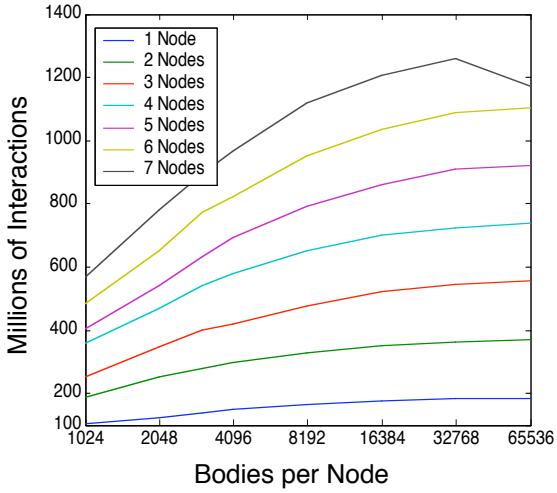


Figure 5: Millions of Interactions vs. Bodies per Node

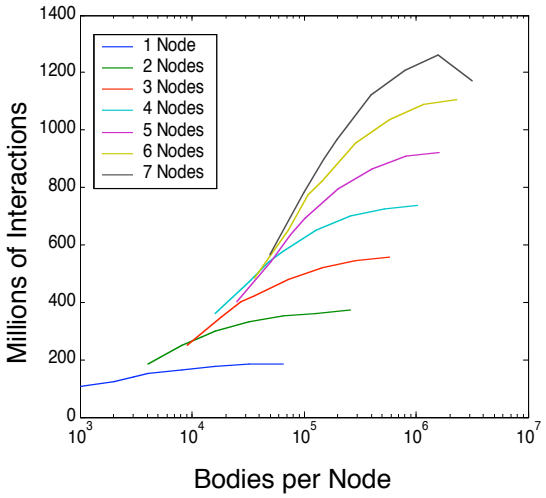


Figure 6: Millions of Interactions vs. Total bodies

7.1 Single-GPU and First Parallel Implementation

Figure 4 shows results for our single-GPU algorithm, our unoptimized parallel algorithm, and a corresponding unoptimized CPU implementation. The CPU implementations shown use the same steps and the same order of computations as our GPU algorithms. We observe that the single-GPU implementation runs at a rate of 71 million interactions per second, which is nearly three times as fast as the corresponding CPU-based algorithm's 24million interactions/second. The CPU implementation scales perfectly to two and three processors, but its interaction rate remains almost the same when a larger number of bodies are used per processor.

Our GPU implementation scales almost identically to the CPU version for both 1024 and 2048 bodies. However, we do see that for the single-GPU the interaction rate is higher for 2048 bodies, while for 2 and 3 processors the opposite is true. This can be attributed to the additional copying overhead for the force accumulation, which was described in §5.

7.2 Optimized Implementation

We ran the optimized n-body implementation discussed in §6 for all node counts from 1 to 7. We started with only node 0, and added node 1, 2, and so on to node 6 for each successive run. For each of these configurations, we varied the number of bodies on each GPU from 1024 to 65,536. The performance results are shown in Figures 5 and 6. We should note that the nodes were ordered in such a way that nodes 1-4 are no slower than node0, but node6 is slower than node5, which in turn is slower than node4 in terms of GPU computational power. We intentionally refrained from using the fastest nodes first, in order to show a smoother speedup as we added nodes to the cluster. Our optimized version runs at 182 million interactions per second when simulating 65,536 bodies on a single node, whereas our seven-node simulation performs as well as 1.26 billion interactions per second. There is a performance decrease when running 65,536 bodies on seven nodes, and this is due to the slower node6.

Since all the nodes have to synchronize at a barrier before exchanging position data, the simulation will only run as fast as the slowest machine. Node 6 is unable to keep up with the other nodes when simulating 65,536 bodies and the entire cluster is forced to wait for it as a result.

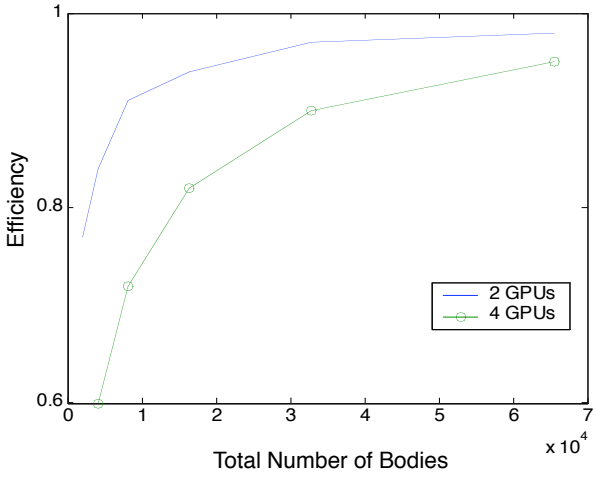


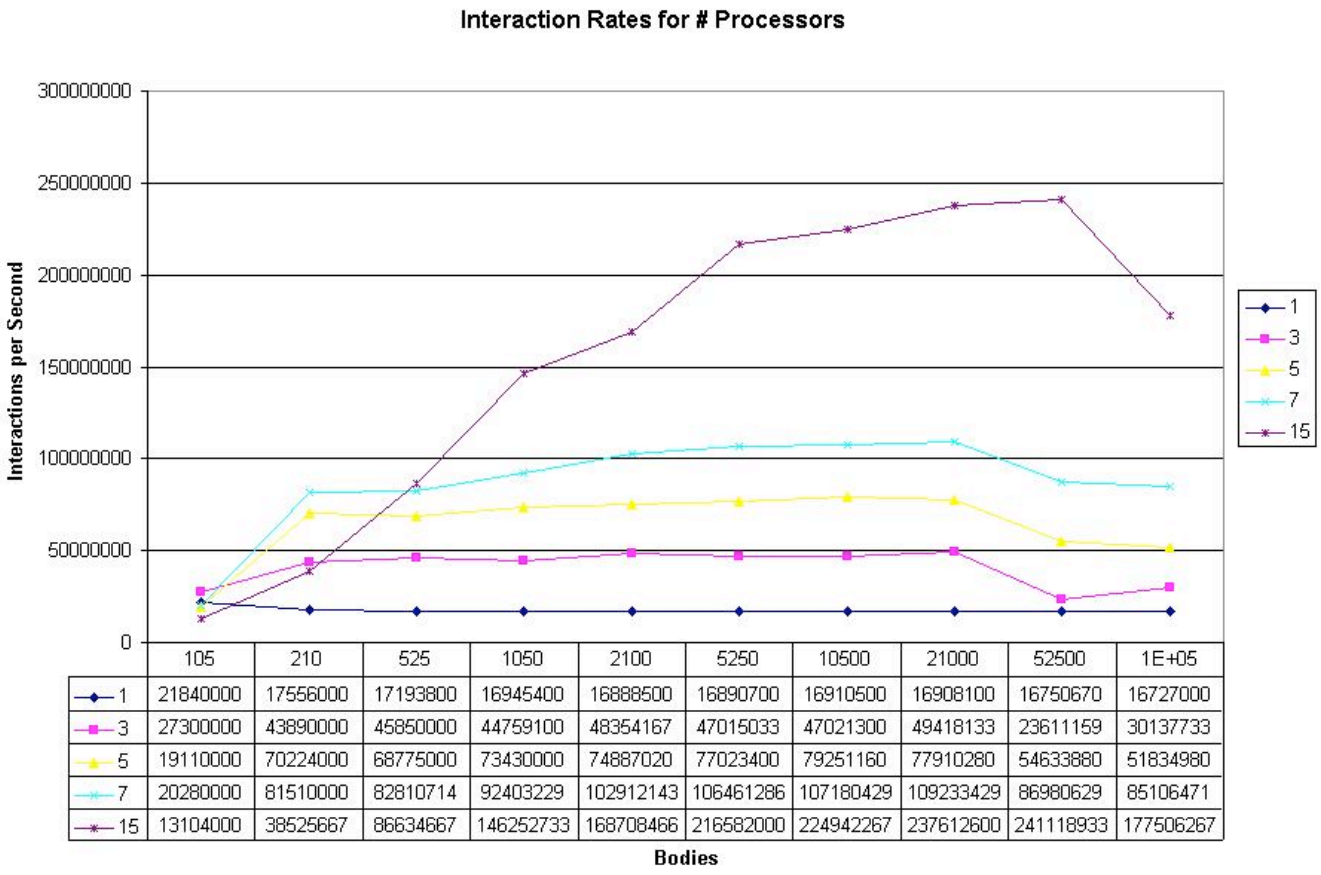
Figure 8: Efficiency of GPU Implementation

Figure 6 shows the same data as Figure 5, only the total number of bodies are used instead of the number of bodies per node.

Finally, we compared our GPU implementation to evans, our department's SGI Origin 2000 system. Figure 7 shows the performance achieved on this machine by [13]. Our single GPU implementation reaches 190 million interactions per second, whereas 15 processors on this system achieve only 125 million interactions per second when performing an equivalent shared memory implementation of the N-body algorithm. Furthermore, Figure 8 shows that the efficiency of our algorithm increases with the number of bodies on each node, so we can reasonably expect far better results with a comparable number of GPUs.

7.3 Divergence of GPU results from CPU

It is well known [3, 8] that N-body algorithms diverge at an exponential rate, and that small errors can balloon incredibly quickly in these systems. The very nature of the force calculation should trigger some degree of alarm in the reader, as it requires us to sum n force values with widely varying



	105	210	525	1050	2100	5250	10500	21000	52500	1E+05
1	21840000	17556000	17193800	16945400	16888500	16890700	16910500	16908100	16750670	16727000
3	27300000	43890000	45850000	44759100	48354167	47015033	47021300	49418133	23611159	30137733
5	19110000	70224000	68775000	73430000	74887020	77023400	79251160	77910280	54633880	51834980
7	20280000	81510000	82810714	92403229	102912143	106461286	107180429	109233429	86980629	85106471
15	13104000	38525667	86634667	146252733	168708466	216582000	224942267	237612600	241118933	177506267

Figure 7: Interaction rates on evans.

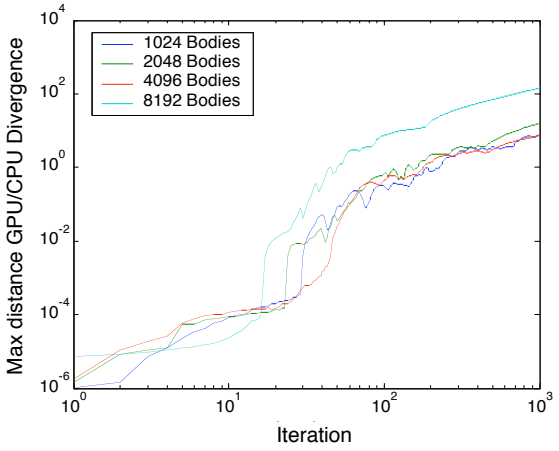


Figure 9: Divergence of GPU from CPU results

magnitudes. Because gravity propagates based on an inverse square law, it is entirely possible in an n-body system to see very small forces incident on bodies from very distant masses, while closer objects exert a much larger amount of pull. The sum of these sorts of floating-point values can easily result in the loss of low-order bits.

Typically when scientists refer to N-body error, it is in terms of *crossing time*. Without delving into unnecessary detail, this is the average time that it takes for any one particle in the system to move from one side of it to the other. In a typical N-body system today, if the relative error per crossing time is 10^{-p}, then after p crossing times, particles in the system will have error equal to its size [3]. Put simply, we cannot know with any accuracy where any particle in the system lies.

Despite these depressing figures, N-body simulation is not valued by cosmologists for its ability to predict precisely the trajectories of individual objects in large systems. Typically, scientists are interested more in the large-scale statistical behavior of the system, e.g. the formation of clusters among bodies, or the spiral motion of a galaxy. It is widely believed (but not proven) that these simulations are valuable and statistically accurate at this scale.

Both to assess the correctness of our algorithm, and to compare floating point error of the GPU to that of the CPU, we computed positions for 1000 .01 second timesteps for both the CPU and GPU implementations. We then compared results at each step.

Figure 9 shows the maximum Euclidean distance between a body's position in the GPU simulation and its position as computed by the CPU at each iteration. We observe that the difference remains very small (less than 10^{-3}) for at least the first 15 iterations, regardless of the number of bodies simulated. We also observe that until approximately 100 iterations, the divergence is less than 1. After this point, however, we can see that the error propagates more rapidly.

We believe that the closeness of our optimized GPU algorithm to the CPU's results through 15 iterations shows that our implementation is correct. The error that we see after this point can be explained in either of two ways:

1. Differences between CPU and GPU floating point implementations. While CPU manufacturers like Intel and AMD are loyal to the IEEE floating point standard, Graphics hardware companies such as nVidia are not committed to compliance. GPU hardware is driven by the game industry and applications in visualization, where speed of implementation is far more important than floating point accuracy or predictability. Furthermore, images that are realistic enough to fool the human eye can be generated with fewer bits of precision than are necessary for most scientific computations.

2. Differences in C and Cg compilers. Our GPU computations are implemented in Cg, a shader language which uses compilers from nVidia. Our CPU implementation, on the other hand, was compiled using Microsoft Visual C++, version 7.1. The optimizations that either of these compilers do (or do not) perform on our code are unknown to us. Given that floating point operations are not commutative, associative, or transitive at a high degree of precision, subtle optimizations in floating point code could result in small perturbations which might lead to very large differences between results of these two codes.

8 Conclusions and Future Work

We have shown that the N-body gravitational simulation can be implemented on the GPU. We have also shown that such algorithms can scale efficiently, even in the presence of limited CPU-GPU bandwidth and high-latency readbacks.

We showed that a system of seven cluster nodes built ad-hoc from commodity parts and consumer graphics hardware can significantly outperform a comparable CPU implementation of N-body. Our algorithm can achieve an interaction rate of 182 million interactions per second, per node, while a CPU implementation running on the latest microprocessors can only attain a rate of 24 million interactions per second, per node.

One conservative measure of FLOPS traditionally used for CPU implementations of the Nbody algorithm states that there are 23 FLOPS per body interaction [13]. Applying this to our GPU implementation yields a rate of 4.37 GFLOPS peak performance. The theoretical maximum performance of the GeForce 6800 is 40 GFLOPS. This shows that even thought we are significantly outperforming CPU implementations of N-body, we are not yet close to utilizing the entire power of the card. We believe that this is due to the poor bandwidth to the first level texture cache, and that our problems are similar to those discussed in [2]. The reference pattern of the N-body force calculation is similar to that of a matrix-matrix multiplication, in that it exhibits little temporal locality and is spread over a large region of memory.

The performance of our cluster could be easily improved by incorporating the following changes:

- Faster memory access in the GPU, for reasons discussed above.
- Faster interconnect between nodes: Our most optimized implementation was run using standard 100baseT connections. We could easily improve latency and throughput by upgrading these to gigabit Ethernet, or to a more advanced interconnect such as Infiniband.
- Faster bus between GPU and CPU: We showed with our first parallel implementation that transfer of data between CPU and GPU can impact performance. Improving this data path can only serve to speed up our algorithm.
- Faster algorithm: The all-pairs algorithm is the most straightforward solution to the N-body problem. Our

performance could be improved by adopting a more sophisticated algorithm. Today's N-body simulation algorithms take into account spatial decomposition and adaptive methods. These algorithms would take considerably more effort to adapt for the GPU, but could yield considerable improvements in performance. We leave these improvements as future work.

References

[1] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of Supercomputing 2004*, Pittsburgh, PA, November 6-12 2004.

[2] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of Graphics Hardware*, 2004.

[3] W. B. Hayes. A brief survey of issues relating to the reliability of simulation of the large gravitational n-body problem. Ph.D. qualifying depth paper, University of Toronto, 1996. Available from: http://www.cs.toronto.edu/~wayne/research/thesis/depth/depth.html.

[4] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 1998.

[5] W. Li, Z. Fan, X. Wei, and A. Kaufman. Gpu-based flow simulation with complex boundaries. Technical Report 031105, Computer Science Department, SUNY at Stony Brook, November 2003.

[6] W. Li, X. Wei, and A. Kaufman. Implementing lattice boltzmann computation on graphics hardware. *The Visual Computer*, 19(7-8):444–456, December 2003.

[7] L. Nyland, M. Harris, and J. Prins. Rapid evaluation of potential fields using programmable graphics hardware. In A. Lastra, M. Lin, and D. Manocha, editors, *GP2, the ACM Workshop on General Purpose Computing on Graphics Hardware*, 2004.

[8] U. of Washington. University of washington high performance computing and communications group [online]. Available from: http://www-hpcc.astro.washington.edu/.

[9] F. Qiu, Y. Zhao, Z. Fan, X. Wei, H. Lorenz, J. Wang, S. Yoakum-Stover, A. Kaufman, and K. Mueller. Dispersion simulation and visualization for urban security. In *Proceedings of the IEEE Conference on Visualization*, Austin, Texas, 2004.

[10] F. Xu and K. Mueller. Towards a unified framework for rapid computed tomography on commodity GPUs. In *IEEE Medical Imaging Conference (MIC)*, Portland, OR, October 2003.

[11] F. Xu and K. Mueller. Ultra-fast 3d filtered backprojection on commodity graphics hardware. In *IEEE International Symposium on Biomedical Imaging (ISBI)*, Arlington, VA, April 2004.

[12] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science*, 52(3):654–663, June 2005.

[13] A. G. Zaferakis, K. Hoff, and C. Weigle. BSP N-body particle system with MPI parallel processing [online]. Spring 2000. Available from: http://www.cs.unc.edu/~andrewz/comp203/hw2.