4-23-2008

Astrophysical System Simulated on Graphics Hardware

Glenn Kavanagh
Trinity University

Astrophysical System Simulated on Graphics Hardware

Glenn Kavanagh

A departmental thesis submitted to the

Department of Computer Science at Trinity University

in partial fulfillment of the requirements for graduation

with departmental honors.

April 23, 2008

Thesis Advisor

Department Chair

Associate Vice President

for

Academic Affairs

Astrophysical System Simulated on Graphics Hardware

Glenn Kavanagh

Abstract

Graphics Processing Units (GPUs) are quickly becoming viable alternatives for certain simulations currently run on CPUs. They are an attractive option because of their inherent parallelism: like older vector-style supercomputers, they perform repeated calculations on large data sets by processing multiple elements at once. Performance is further improved by their built-in mathematical functions. This paper describes experiments comparing the performance of a GPU with that of a CPU for an astrophysical simulation, using NVIDIA's CUDA programming interface for the GPU.

Acknowledgments

The author would like to thank Dr. Mark Lewis for his boundless efforts to make undergraduate research possible to anyone who has a drive to do so and especially for his help on this project, Dr. Berna Massingill for her efforts to install drivers and fix system problems, as well as her LaTeX knowledge, Dr. Maurice Eggen for providing excellent feedback and corrections, and the Department of Computer Science at Trinity University for purchasing the hardware to make this research possible.

Astrophysical System Simulated on Graphics Hardware

Glenn Kavanagh

Contents

List of Figures

Chapter 1

Introduction

Moore's Law has forced the major CPU manufacturers away from developing single processor CPUs; instead they are now focusing on multi-processor CPUs. Only with the inclusion of four distinct cores has the software industry begun looking at developing software that is multi-threaded. This new focus in parallel application development has sparked new interest in previous work on parallel application development, such as the pattern language PLPP [10]. Much of this work has focused on multi-CPUs platforms such as multicore chips and clusters. The Graphics Processing Unit (GPU), however, is beginning to emerge as a viable option for use in certain types of computations, including scientific simulations. Such use is referred to as general purpose computing on graphics hardware (GPGPU). There is a fundamental difference between the parallelism gained from multiple CPUs and a single GPU [9]. GPUs create parallelism through stream processing. That is, given an arbitrarily sized data set, the GPU can apply the same function to n elements at a time, where n is the number of pipelines on the GPU. This is, in many ways, similar to older vector-style supercomputers. Current high-end gaming graphics cards, as well as high-end workstation graphics cards, have 16 pipelines, and they have between 512MB and 1.5GB of on-board RAM. The large amount of RAM allows the GPU to hold the data for the computation on the card, which decreases memory latency issues between system RAM and the GPU. Along with the inherent parallelism of stream processing, the GPU was built to do calculations quickly to increase the video quality and refresh rate for video output, which also contributes to its suitability as a platform for intensive numerical calculations.

The most significant change in the last two generations of GPUs from the standpoint of GPGPU is increased programmability. Earlier GPUs could only perform the required graphics operations, while the newer GPUs allow program logic for applications such as vector/pixel shaders. Future generations of GPUs are likely to offer even more programmability, along with overall increases in speed, and enhancements such as support for double-precision floating-point numbers. Chip makers such as AMD and Intel are also considering adding vector-processing units similar to those in GPUs to CPUs. This is most apparent in the AMD Fusion project.

The built-in parallelism and increased calculation speed make the GPU an acceptable platform for performing certain types of simulations, specifically those that are data-parallel. (Data-parallelism, in which parallelism is obtained from operating concurrently on elements of a large data structure, has a long history, going back to vector-style supercomputers and some early massively-parallel platforms such as the original Connection Machine [6]. Some early and interesting examples are given in [7].)

The goal of the work presented in this paper is to compare, for one such simulation, the performance of a CPU-based implementation and its GPU equivalent. For this purpose, an astrophysical simulation of a simple planetary system using a simple symplectic $T + V$ integrator proved to be ideal.

1.1 GPGPU: A Brief History

Research into graphics rendering has been around for nearly three decades (at the time of this writing). Since 1978, machines such as the Ikonas [3], the Pixel Machine [12], and Pixel-Planes 5 [14] have been built and used for this purpose. However, these computers do rendering and image processing, and more importantly, these devices were not readily available to the public.

With the advent of video cards for personal computers, the possibility of doing general purpose computation on graphics hardware became more feasible. However, there was still a major stepping stone yet to cross.

1.1.1 GPUs for the Masses

Even as GPUs began to be developed by companies such as 3dfx, ATI, and Matrox, most research was still focused towards improving the image processing features of the graphics hardware. Prime examples of this type of research can be seen in papers authored by Wolfgang Heidrich. He presented methods to implement wider varieties of reflection models using multi-pass methods on the hardware he had at his disposal [4], as well as more flexible parabolic parametrization of environment maps [5] which allowed for multiple viewing directions. These advances did start the ball rolling on the idea that a much broader range of calculations could be done on the GPU. The issue persists that most of this "early" research dealt exclusively with the visualization rather than the calculation.

Chris Trendall and James Stewart go into a more in-depth background on what graphics hardware was capable of eight years ago in [15]. In the same paper, they illustrate one of the big problems with early attempts at general processing on graphics hardware: conforming to the tools you have. They outline how functions must be scaled and biased to fit into the [0,1] range, which can then be stored as a vector of the red-green-blue(-alpha) colors. This roundabout way of doing things prevented many researchers from trying to start using the full potential of the GPU.

Sometime around 2003, the Stanford University Graphics Lab released a beta version of BrookGPU [8] (briefly described in Section 2.2). An extension to C, it provided a DirectX and OpenGL back-end allowing programmers to start utilizing the full potential of GPUs.

Finally, in 2006 and 2007, the two major GPU manufactures (AMD/ATI and NVIDIA) released their own GPGPU languages (CTM [1] and CUDA [11] respectively). These new languages allow programmers a previously unavailable ability to access the graphics hardware and use it like a CPU.

Chapter 2

GPGPU Technologies

There are several technologies currently available to GPGPU enthusiasts. The following discusses some of these.

2.1 NVIDIA CUDA

For this experiment into the performance advantages of the GPU, NVIDIA's Compute Unified Device Architecture (CUDA) was used. CUDA [11] is a combination hardware/software implementation that provides the user the ability to issue and manage computations on the GPU without the need to use a graphics API such as OpenGL or Microsoft's DirectX. This allows programmers unfamiliar with these APIs to bypass any learning curve associated with this step and write their code in a more natural style; the learning curve is further reduced by the fact that the CUDA API is an extension of a widely-known language, C. Currently, CUDA is available on the GeForce 8 Series, the Tesla solutions, and several Quadro solutions. Results presented in this paper were obtained using an NVIDIA GeForce 8800, which utilizes

16 separate SIMD multiprocessors. An interesting note about the Tesla solutions mentioned: they are specifically designed for GPGPU and come in a variety of different configurations including single GPU cards, a 2 GPU desk-side system, or a 1U server containing 4 GPUs.

Using CUDA, the GPU can be viewed as a compute device that can execute many threads in parallel. This allows for portions of programs that are compute-intensive and data-parallel, such as the simulation described in this paper, to be offloaded onto the GPU (device) from the CPU (host).

Because the GPU has its own memory that can be used for calculations, the CUDA API provides functions to copy data from the host memory into the device's global memory and back into the host memory. The CUDA memory model is discussed in more detail later.

2.1.1 Threading

An easy way to look at the CUDA threading model is to describe it as a grid of thread blocks, where kernels (code to be executed on the device) are executed on a grid (visually illustrated in Figure 2.1). That is, each thread is in a block, and all threads in that block can communicate with each other and have access to the same shared memory space (discussed in Section 2.1.2). Each of these blocks is executed on a separate SIMD multiprocessor. This means that threads in different blocks do not have access to the same shared memory. It is possible to execute multiple device functions on the different multiprocessors that are not associated with the same problem — or more precisely, multiple GPU applications can be executed simultaneously, and each one will be processed on a different multiprocessor.

To help coordinate between threads, CUDA provides a `__syncthreads()` func-

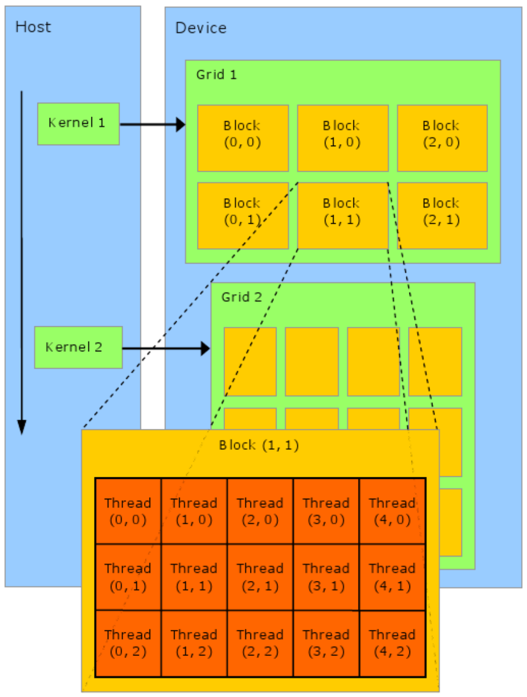

Figure 2.1: NVIDIA CUDA threading model from [11]

tion which acts as a barrier. Once all threads being executed in the kernel have reached that point, they are allowed to continue. However, the `__syncthreads()` function only works on threads within the block it is called in — that is, kernels that are executed over multiple blocks cannot implement algorithms that require synchronization over all threads. It is this reason that for this research, only a single block (multiprocessor) is utilized.

2.1.2 Memory Model

It is easiest to view the CUDA memory model as three-tiered, with the top tier consisting of global memory, constant memory, and texture memory. All three of these can be written to and read from by the host. Both the constant memory and texture memory are cached and are read-only per grid. Global memory is not cached (leading to problems detailed below) but is read-write per grid. Shared memory is located in the second tier. It is read-write per block and provides a way to share memory between the different threads contained in that block. The third tier contains registers and local memory. Both of these are read-write per thread.

Strictly speaking, access to global memory takes a great deal of time, approximately 400 to 600 clock cycles to read a `float` from global memory [11]. The solution to this problem is the on-chip shared memory for each multiprocessor (briefly described in Section 2.1.1). A read from shared memory takes 4 clock cycles, and a write takes 4 clock cycles. However, shared memory is limited in size with 16KB, while the global memory space can utilize 512MB. Because of this, one of the main challenges was determining how to use shared memory as often as possible. This is described in Section 3.3.4.

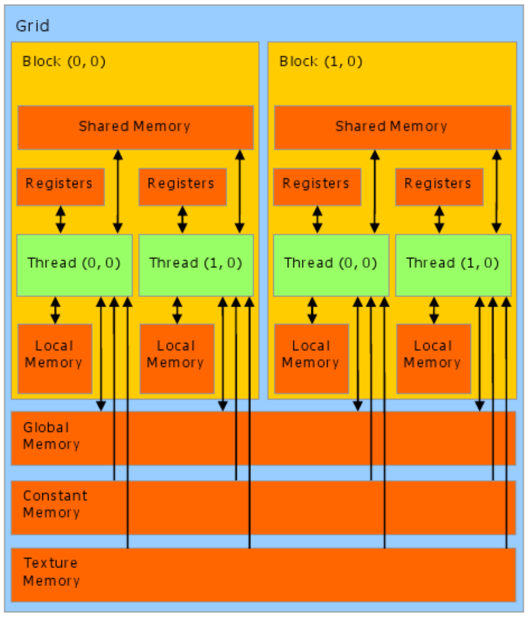

Figure 2.2: NVIDIA CUDA memory model from [11]

2.2 Other Technologies

There are several other programming environments for developing GPGPU programs. Most notable is AMD/ATI's competitor to CUDA, Close To Metal (CTM) [1]. The approach taken by ATI differs from NVIDIA CUDA in that CTM allows the programmer to write directly in assembly language. The hope is that a community will form around CTM that will provide higher-level tools, whereas CUDA forces users into using their implementation.

BrookGPU [8] is another alternative to CUDA that has been in development for several years. Recent development has focused on using CTM as a back end. BrookGPU is based on the Brook stream processing language [13]. It provides different back-end implementations for DirectX 9, OpenGL, and now ATI's CTM.

For this work, CUDA was chosen for several reasons. First, it already provided a high-level API and tools to use, in contrast with CTM. Also, the hardware was available for purchase at the right time. That is, CTM-compatible cards were not readily available, but the NVIDIA cards that were CUDA-compatible were.

Chapter 3

From C to Stream

The first approach to adapting an astrophysical simulation from the CPU onto the GPU was to decide on which integrator to use. For this work, a standard $T + V$ symplectic integrator was chosen. This style of integrator has the benefit of simplicity while still being accurate enough to be useful to researchers.

3.1 Numerical Integrator

The actual numerical code used for this project is an implementation of a first-order T+V symplectic integrator. This type of integrator is not uncommon in simulations of astrophysical systems. Being symplectic means that this type of integrator has certain nice properties when integrating Hamiltonian systems [2]. Because gravitational systems are energy conserving Hamiltonian systems this is very beneficial. The code itself is not much harder to write than a first-order Euler method, but while an Euler method will quickly gain energy in this type of system, the symplectic integrator won't. This is because symplectic integrators, also known as symplectic maps, are volume conserving maps and for a Hamiltonian system they not only preserve energy, but also phase space volumes, and the Poincar invariants of the system. They do this because they perform exact solutions to systems with slightly different Hamiltonians than the one being integrated. As long as this so-called perturbed Hamiltonian is bound to the actual Hamiltonian, even low-order symplectic maps can be much more accurate for long-term integrations than much higher order standard methods.

The T+V in the name comes from the style of the integrator. As it happens, the composite of two symplectic maps is itself symplectic and a map built from an integrable system is always symplectic. So the T+V style integrators break the Hamiltonian into two integrable pieces, one for the kinetic energy and one for the potential energy. When these are performed one after the other, the result is a symplectic mapping for the whole system. This integrator can be extended to second order by simply taking a half step in the first and last steps of the simulation as the midpoint of each step is second order accurate.

3.2 CPU Version

Pseudocode for the CPU version is shown in Figure 3.1. There are two basic steps to this code. First, accelerations on the particles are calculated and used to update the particle velocities. Then the updated velocities are used to change the particle locations. The majority of the work happens in the calculation of the accelerations, which is $\Theta(n^2)$. The nested loops here must run through all pairs of particles. To reduce the overall workload, the distance between each pair is only calculated once and accelerations are calculated for both particles. One can visualize the set of interacting pairs as a 2D grid across `i` and `j`. This implementation runs through the upper left triangle of such a grid.

```
for n=1 to num_steps
    for i=0 to num_particles
        for j=i+1 to num_particles
            calculate velocity for particles[i];
            calculate velocity for particles[j];
    for i=0 to num_particles
        adjust position for particles[i];
```

Figure 3.1: Pseudocode for the CPU and Triangle versions of the simulation.

3.3 GPU Code

In adapting the CPU code to GPU code, the flow of the program was altered to fit into the GPU programming paradigm. That is, the code was split into host functions and device functions. Within these distinctions lies the essential difference between writing code that will run on a CPU and that to be run on a GPU.

First, there is C code to be executed on the host, consisting of a main function. Within this main function there is a call to a device function, or a call to another host function that eventually calls a device function. This device function contains the code that is to be run on the device (GPU).

In this simulation, host functions are used to create the particles, store them in a `struct`, and calculate energy, which is used to verify the conservation of energy after a simulation is finished running. The actual integration — being the bulk of the calculation — is performed in a device function.

3.3.1 Initial Test

As a first test, the C code was copied almost verbatim into the new GPU code base. Following the programming paradigm described in Section 3.3, the newly created GPU code used global memory. This, as implied in Section 2.1.2, led to a sub-optimal result. When run times were measured and compared to the results from the C code, the GPU ran significantly slower. To improve performance, several versions of the code were developed. The first of these was the Triangle version.

3.3.2 Triangle

The Triangle version — named after the geometric shape resembling how the integrator evaluates on `i,j` pairs, as shown in Figure 3.2 — is almost exactly like the initial test of the GPU code. The only difference is the modification of the kernel code. Before any steps are executed on the device, the particles are loaded into the shared memory space to take advantage of lower memory-access times.

3.3.3 Square

The Square version of the code follows the pseudocode shown in Figure 3.3. This version of the code calculates distances for all `i,j` pairs instead of just the upper left triangle. The initial idea behind using a square configuration was that it would perhaps be better for the stream-style processing of the GPU if all of the threads had an equal number of particles to evaluate, so that all threads would always be working until it finished. The only possible snag for this approach was the addition of an `if`-statement, which could produce some branch-predicting problems. That is, the nature of stream processing does not work well with branching because it

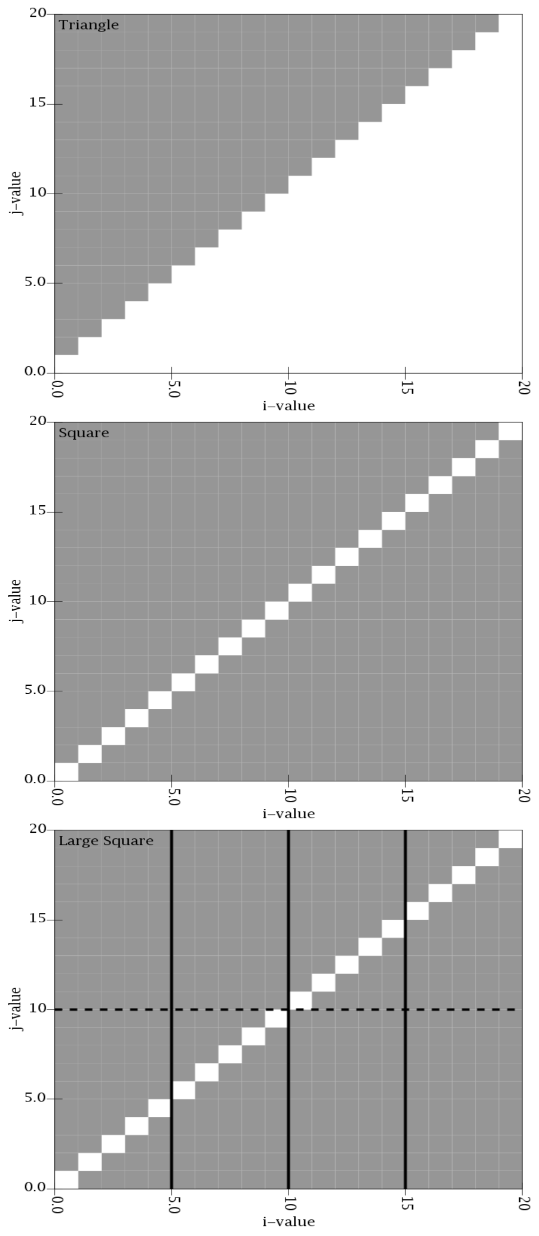

Figure 3.2: Illustrations of the execution styles for each of the three different GPU code versions. Gray signifies an `i,j` pair that the version executes on; white means no work. For the Large Square figure (third), the solid lines represent the grouping of active threads, and the dashed line represents the grouping of chunks.

breaks the idea that one discrete function should be applied to the whole data set. Fortunately, NVIDIA allows branching with minimal overhead as long as there are less than seven instructions inside the `if`-statement. While the first version of the code did not meet this condition, it was possible to refactor it by moving the temporary variable assignments outside

```
for k=1 to num_steps
    for i=0 to num_particles
        for j=0 to num_particles
             if i!=j
                 calculate velocity for particles[i];
    for i=0 to num_particles
        adjust position for particles[i];
```

Figure 3.3: Pseudocode for the Square version of the simulation.

A more significant problem arose with simulations with larger data sets. Any simulation with more than 512 particles threw an error stating that there was not enough shared memory. The size of shared memory, 16KB, was not enough for the increased size of the data sets. To get around this, a new path was taken.

3.3.4 Large Square

The increase in data set size forced the computation to be divided in a different way. In order for there to be a reasonable prospect of the GPU code being faster than the CPU, it would need to put as many of the particles into the shared memory space as possible. To do this, a system was devised to “chunk” the particles into manageable sections that would fit in shared memory.

In the $T+V$ integrator used for this experiment, first the velocities are altered for all particles, and then the positions are changed. As in the smaller simulations, the velocity calculations are fully completed before the position calculations. The size of the data set became a problem in the velocity calculations. To get around this, the solution was built around the Square version of the GPU code. This version was chosen over the Triangle version simply because the ease of chunking in the Square version was more intuitive.

For each thread, the current particle positions and velocities for those particles were stored locally. A new `for`-loop was added to deal with the chunking. Each iteration of this loop loads a portion of the particles into the shared memory space from global memory. The chosen size of each chunk was 256, which corresponds to the number of particles that can be kept in shared memory at any one time. After this, the change in velocity due to the particle in the current chunk are calculated. Once all threads finish — enforced by a `__syncthreads()` call — a new chunk is loaded into shared memory. Once the velocities are applied to the currently held local particles, those are written back out to global memory, and new particles loaded in. Following the completion of the velocity calculations, the particle positions are calculated normally.

Modified Data Structure

With the new chunking scheme came a need to hold the particle positions and their velocities separately. For this, the built-in `float4` data-type was used. The `float4` holds four floats, which can be accessed through the public data members `x,y,z,w`. For the positions, the `w` data member was used to hold the mass. For the velocities, the `w` was not used.

3.4 Performance

The original CPU code and the different versions of the GPU code were run for different numbers of particles and different numbers of threads (for the GPU). Timing results were taken and are presented below.

Figure 3.4 represents the baseline CPU version of the code. The $T+V$ integrator is $\Theta(n^2)$ as can be seen here. An interesting occurrence on the CPU graph is the substantial jump in time taken between 128 particles and 256 particles. This is believed to be due in part to the L1 cache size on the processor. Once the number of particles rises to 256, they can no longer be stored in the L1 cache alone. However, the L2 cache is sufficient to hold the larger data sets.

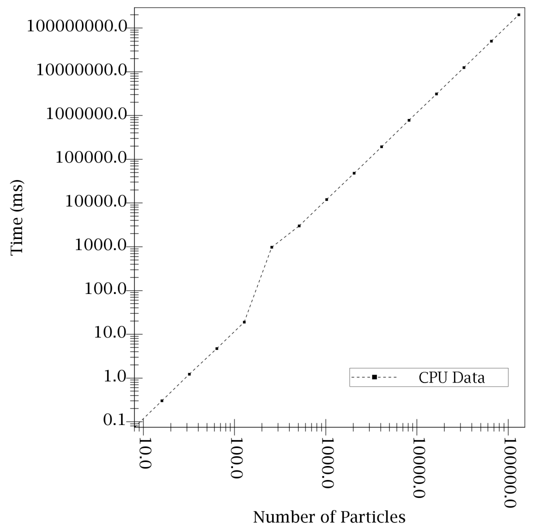

Figure 3.4: Timing results for the CPU code for particle counts ranging from 8 up to 131072 (256 × 512.)

Figures 3.5 and 3.6 show the timing results for the Square and Triangle data respectively. Overlaid on each graph are the timing results for CPU version (the dashed lines) for comparison. For each of the different thread counts, the lowest number of particles tested was 8, and higher thread counts were not tested on particle counts lower than their thread count.

Each dot represents a different timing result, with the dot size corresponding to a different number of threads. From top to bottom, the number of threads increases. They increase by powers of two, where the uppermost line represents one thread and the single point below the last line (barely visible) represents 512 threads. This small detail is noteworthy as the jump from 256 threads to 512 threads sees very little improvement, though there is some.

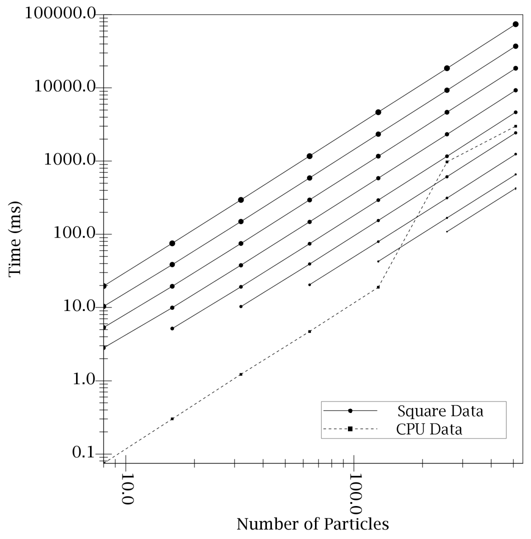

Figure 3.5: Timing results for the Square version of the GPU code; particle count ranges from 8 to 512, and thread count ranges from 1 to 512 (each range increasing by powers of 2).

These graphs show a clear delineation of where the GPU becomes more efficient to use than the CPU. For the Square version of the code, this can be seen when the particle count moves from 128 to 256, and the number of threads changes from 16

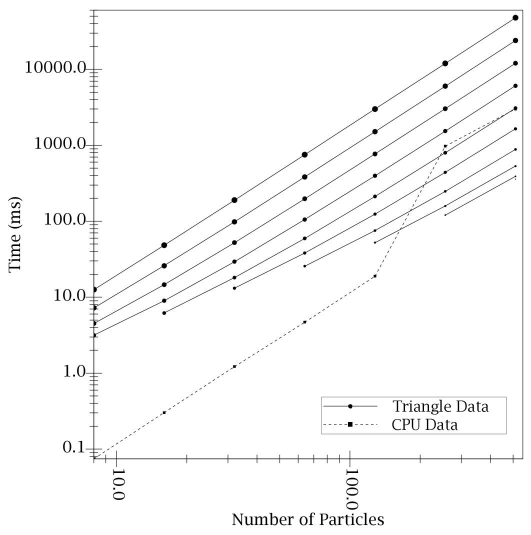

Figure 3.6: Timing results for the Triangle version of the GPU code. Note in Figure 3.5 applies here as well.

to 32. When using 512 threads on 512 particles, the GPU completes the simulation in 411 milliseconds (ms), while the CPU took 3001 ms — approximately 7.3 times slower.

Comparatively, the Triangle version of the code runs slightly faster than the Square version — 359 ms to 411 ms using 512 threads. This increased speed can be seen in Figure 3.6, where the CPU is beaten initially by fewer threads, 16, following the increase in the number of particles to 256. The Triangle version, however, cannot sustain the lead with 16 threads. At 512 particles, at least 32 threads should be used to take advantage of the GPU. This is consistent with the Square version.

Figure 3.7 shows the timing results from the Large Square version of the GPU code. Here, the different lines represent different thread counts increasing by powers of 2 from 16 up to 256, with the dashed line still representing times for the CPU code. Figure 3.7 paints a slightly different picture than Figures 3.5 and 3.6. The Large Square takes more time to complete a simulation than the smaller simulations. This occurs because of the increase in complexity required to use more particles, and also because of the additional time needed to read and write to and from global memory. For this reason, a thread count of 256 must be used to be faster than the CPU code. Though the speed increase is only a factor of 1.55, this is still significant for large-scale simulations; using a particle count of 131,072, the CPU took 55.86 hours, while the GPU using 256 threads took 36.07 hours.

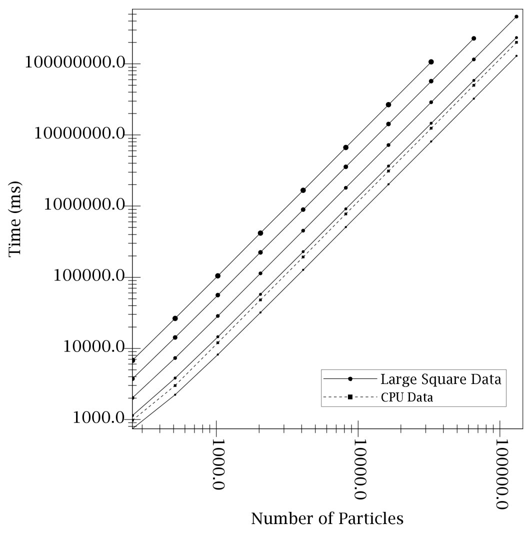

Figure 3.7: Timing results for the Large Square version of the GPU code.

Chapter 4

Conclusions and Further Discussion

4.1 Conclusion

The results in the preceding section show a definite increase in speed when the GPU is used. Depending on the size of the simulation running, the lower thread count results can be overlooked because the number of particles present would exceed the number of threads. More importantly, the results produced on the small simulations are relatively inconsequential as far as time spent is concerned. In these cases, the effort expended to run the simulation on the GPU could better be spent analyzing the results produced by the CPU. It could also be postulated that because the CPU is no longer working on the simulation, it would be free to work on other jobs, such as a different simulation.

As briefly mentioned in Section 2.1, device functions can be run on multiple thread blocks. This research used only one thread block — multiprocessor — because of the need for synchronization between threads. Assuming no need for this synchronization, more threads could be used. In this case, multiple simulations could be run at the same time using the separate multiprocessors, thus allowing even more work to be accomplished in parallel.

4.2 Suggestions for Further Research

The results of this research were favorable. However, there are several possible improvements that could be made, as well as other avenues to fully exploit the power of the GPU as a compute device.

4.2.1 Implementing a Tree

When dealing with any application development, it is usually necessary to store state information in some data structure. These can range from a simple unordered array to doubly-linked lists to AVL trees. As programs grow in complexity, they tend to require more advanced data structures to hold the information. Astrophysical simulations are no different. For low particle-count simulations, a flat array is just fine. However, once the simulations begin to grow, it can be advantageous to implement a tree data structure. In fact, a kD-tree allows force calculations to be evaluated in $\Theta(n \log n)$ time. Because of overhead, this only provides a speedup once the particle count reaches the 1000 to 10,000 mark. As with most advanced data structures, there is a lower bound where the complexity of the structure is a disadvantage.

When implementing a kD-tree for the simulation described here, several changes occur in the basic C code, the largest of which is building the tree, which must be done at each timestep. Also, more logic must be added to the particle acceleration updates to traverse the tree. Of course, this is where a speed increase is seen in the C code because the velocity calculation no longer requires a comparison to each particle. Instead, the traversal can stop at a non-leaf node n if it is determined that the nodes under n cumulatively act against the current particle.

Several problems may occur when trying to implement a kD-tree on the GPU. First is that the chunking algorithm described in Section 3.3.4 will not work with the tree. A possible solution to this may be to hold onto an array of the indexes of the particles in shared memory and the actual particles can reside in global memory. This would allow the finding of the particles to happen quickly and the reads from global memory to only occur when a particular particle is needed.

Another problem may occur with the actual building of the tree. Because of the recursive nature of this operation, as well as the need to sort around a pivot, this might have to be done with only one thread, in which case any speedup would have to occur in the velocity calculations. Of course, the actual sorting could be done with a parallel sorting algorithm, which would just need to be followed by a thread synchronization prior to the successive recursive call. (In fact, there is a sort implemented in the CUDA sample programs.)

Any implementation would need to be tested against both the CPU results and the flat array results.

4.2.2 Implementing a Queue

One initial idea for this research was to essentially create a queue that threads could then pull work off of when they were not busy. Assuming something like this could be efficiently implemented, it would open up the possibilities to many different data structures serving as a back-end to the simulation. Also, taking the code base developed for this idea, creating different simulations would be easier because they could just pass work off to the threads. One possible problem, however, would be that it might break the stream processing speedups. This would perhaps be more useful for calculations that could be done across more than one of the SIMD multiprocessors.

4.2.3 Different Integrator

One problem with using a simple integrator is that the computational complexity for any `i,j` pair of particles is very low. That is, the integrator is simple enough that the CPU version should be reasonably fast. It could then be reasoned that the GPU is not being given enough work to produce the highest possible speed benefits. If a more complex integrator was swapped in, the CPU would see an increase in processing time, and the GPU's processing time could increase at a slower rate.

4.2.4 Double-precision

In the next few generations of GPUs, both AMD/ATI and NVIDIA have said they will introduce the ability to use double-precision floating point numbers on their respective hardwares. It would be interesting to determine if the performance results remain constant on the new hardware. This is essential for most scientific work because of the need for higher precision in simulations.

4.2.5 Distributed GPGPU and Multi-block Simulations

One of the more fascinating accomplishments of the drive for increased parallel computing is that of distributed simulations — those utilizing multiple physical computers that communicate with each other by passing messages over the network. Using this technique, more research could be done to determine if there is any performance benefit to doing this, or if the more frequent reading/writing to global memory would make this a poor option.

This goes hand in hand with experiments in synchronizing threads across multiple blocks (multiprocessors). If a suitable way to accomplish this was discovered — or eventually implemented in CUDA — the performance results would be interesting to analyze.

4.3 Concluding Remarks

Having briefly interacted with DirectX code in the past and after reading several papers that had OpenGL example code, CUDA felt like a cool breeze on a hot summer's day, at least in the sense of understanding and being able to use the syntax it sets forth. Adapting C code to (inefficient) CUDA code is as simple as adding a `__global__` modifier to the beginning of a function definition.

From this research, however, writing CUDA code well relies heavily on knowing the intricacies of the architecture, which can be said for many languages and APIs. Of course, knowing where to store values in memory and remembering to keep `if`-statements under seven instructions are not the only problems encountered. Like any other tool, GPGPU should only be used to solve certain problems. Java isn't used to write console-based scripts, just as Perl isn't used to write GUI applications. The GPU has already begun to carve out its niche as a viable option to perform data-parallel simulations.

Bibliography

[1] AMD. *ATI CTM Guide*, 2006. Version 1.01.

[2] P. Channell and C. Scovel. Symplectic integration of Hamiltonian systems. *Nonlinearity*, 3.

[3] J. Nick England. A system for interactive modeling of physical curved surface objects. In *Proceedings of SIGGRAPH 78*, pages 336–340, 1978.

[4] Wolfgang Heidrich and Hans-Peter Seidel. Efficient rendering of anisotopic surfaces using computer graphics hardware. In *Proceedings of the Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP)*, 1998.

[5] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39–45, 1998.

[6] W. Daniel Hillis. *The Connection Machine*. MIT Press, 1985.

[7] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[8] Stanford University Graphics Lab. BrookGPU. `http://graphics.stanford.edu/projects/brookgpu`.

[9] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. SIMD: An additional pattern for PLPP (Pattern Language for Parallel Programming). In *Proceedings of the Fourteenth Pattern Languages of Programs Workshop (PLoP 2007)*, September 2007.

[10] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[11] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*, June 2007. Version 1.

[12] M. Potmesil and E.M. Hoffert. The pixel machine: a parallel image computer. In *Proceedings of SIGGRAPH 89*, pages 69–78, 1989.

[13] Merrimac-Stanford Streaming Supercomputer Project. Brook. `http://merrimac.stanford.edu/brook`.

[14] J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney. Real-time procedural textures. In *Proceedings of Symposium on Interactive 3D Graphics 1992*, pages 95–100, 1992.

[15] Chris Trendall and A. James Stewart. General calculations using graphics hardware, with applications to interactive caustics. In *Eurographics Workshop on Rendering 2000*, 2000.

Appendix A

Code

A.1 nbody.c

Initial C code (described in Section 3.2).

```
/*
 *  Originally written by Christopher Bauer
 *    for
 *    The Great Computer Language Shootout
 *    http://shoutout.alioth.debian.org/
 *
 *  Altered for of the use of this
 *    research.  Specifically, changed the
 *    way the planets/particles are stored.
 *    Also, uses more than the initial five
 *    planets.
 *
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define pi 3.141592653589793
#define solar_mass 1
```

```
#define days_per_year 365.24

struct planet {
   float x, y, z;
   float vx, vy, vz;
   float mass;
};

struct float4 {
   float x,y,z,w;
};

void advance(int nbodies, struct float4 * pos, struct float4 * vel, float dt)
{
   int i, j;

   for (i = 0; i < nbodies; i++) {
      for (j = i + 1; j < nbodies; j++) {
         float dx = pos[i].x - pos[j].x ;
         float dy = pos[i].y - pos[j].y;
         float dz = pos[i].z - pos[j].z;
         float distance = sqrt(dx * dx + dy * dy + dz * dz);
         float mag = dt / (distance * distance * distance);
         vel[i].x -= dx * pos[j].w * mag;
         vel[i].y -= dy * pos[j].w * mag;
         vel[i].z -= dz * pos[j].w * mag;
         vel[j].x += dx * pos[i].w * mag;
         vel[j].y += dy * pos[i].w * mag;
         vel[j].z += dz * pos[i].w * mag;
      }
   }
   for (i = 0; i < nbodies; i++) {
      pos[i].x += dt * vel[i].x;
      pos[i].y += dt * vel[i].y;
      pos[i].z += dt * vel[i].z;
   }
}

float energy(int nbodies, struct float4 * pos, struct float4 * vel)
{
   float e;
   int i, j;

   e = 0.0;
   for (i = 0; i < nbodies; i++) {
      e += 0.5 * pos[i].w * (vel[i].x * vel[i].x
         + vel[i].y * vel[i].y + vel[i].z * vel[i].z);
      for (j = i + 1; j < nbodies; j++) {
         float dx = pos[i].x - pos[j].x;
         float dy = pos[i].y - pos[j].y;
         float dz = pos[i].z - pos[j].z;
         float distance = sqrt(dx * dx + dy * dy + dz * dz);
         e -= (pos[i].w * pos[j].w) / distance;
      }
   }
   return e;
}

void offset_momentum(int nbodies, struct float4 * pos, struct float4 * bodies)
{
   float px = 0.0, py = 0.0, pz = 0.0;
   int i;
   for (i = 0; i < nbodies; i++) {
      px += bodies[i].x * pos[i].w;   // pos.w = mass
      py += bodies[i].y * pos[i].w;
      pz += bodies[i].z * pos[i].w;
   }
   bodies[0].x = - px / solar_mass;
   bodies[0].y = - py / solar_mass;
   bodies[0].z = - pz / solar_mass;
}

void create_particles(int len, struct float4 * pos, struct float4 * vel)
{
   int i;
   pos[0].x=0;
   pos[0].y=0;
   pos[0].z=0;
   vel[0].x=0;
   vel[0].y=0;
   vel[0].z=0;
   pos[0].w=solar_mass;
   for (i=1; i<len; i++) {
      pos[i].x=i;
      pos[i].y=0;
      pos[i].z=0;
      vel[i].x=0;
      vel[i].y=1/sqrt(i);
      vel[i].z=0;
      pos[i].w=1e-4;
   }
}


int main(int argc, char ** argv)
{
   int n=628;
   int i;
   int num_p=NUM_P;

   struct timespec start,end;

   struct float4 pos[num_p];
   struct float4 vel[num_p];
   create_particles(num_p,pos,vel);

   offset_momentum(num_p, pos, vel);
   printf ("%.9e\n", energy(num_p, pos, vel));

   clock_gettime(CLOCK_REALTIME,&start);
   for (i = 1; i <= n; i++)
      advance(num_p, pos, vel, 0.01);
   clock_gettime(CLOCK_REALTIME,&end);
   printf ("%.9e\n", energy(num_p, pos, vel));
   double e=((double)end.tv_sec) + ((double)end.tv_nsec/10e9);
   double s=((double)start.tv_sec) + ((double)start.tv_nsec/10e9);
   double time=(e-s)*1000;
   printf ("Processing time: %f\n",time );
   FILE *fout=fopen("times.txt","at");
   if(fout==NULL) fout=fopen("times.txt","wt");
   fprintf(fout,"%d %f\n",NUM_P,time);
   return 0;
}
```

A.2 Main.cu

The following represents a generic entry point into the GPU code execution.

```
// includes, system
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <math.h>

// includes, project
#include <cutil.h>

// includes, kernels
#include <n-body-square-small_kernel.cu>


////////////////////////////////////////////////////////////////////////////////
// functions


void doAdvance(int nSteps, int nBodies, struct planet * parts, double dt)
{
   CUT_DEVICE_INIT();
   int i;
   struct pass bodies[1];
   for (i=0; i<nBodies; i++) {
      bodies->p[i].x=parts[i].x;
      bodies->p[i].y=parts[i].y;
      bodies->p[i].z=parts[i].z;
      bodies->p[i].vx=parts[i].vx;
      bodies->p[i].vy=parts[i].vy;
      bodies->p[i].vz=parts[i].vz;
      bodies->p[i].mass=parts[i].mass;
   }

   struct pass * d_Planets;
   int mem_size=sizeof(pass);


   CUDA_SAFE_CALL(cudaMalloc((void**) &d_Planets, mem_size));

   // copy host memory to device
   CUDA_SAFE_CALL(cudaMemcpy(d_Planets, bodies, mem_size,
            cudaMemcpyHostToDevice));

   // create and start timer
   unsigned int timer = 0;
   CUT_SAFE_CALL(cutCreateTimer(&timer));
   CUT_SAFE_CALL(cutStartTimer(timer));

   // execute kernel
   kernel_advance<<< 1, BLOCK_SIZE >>>(nBodies, d_Planets, dt, nSteps);

   // check if kernel execution generated and error
   CUT_CHECK_ERROR("Kernel execution failed");

   // copy result from device to host
   CUDA_SAFE_CALL(cudaMemcpy(bodies, d_Planets, mem_size,
            cudaMemcpyDeviceToHost));

   // stop and destroy timer
   CUT_SAFE_CALL(cutStopTimer(timer));

   // write timing result out to file
   double time=cutGetTimerValue(timer);
   printf("Processing time: %f (ms) \n", time);
   FILE *fout=fopen("times.txt","at");
   if(fout==NULL) fout=fopen("times.txt","wt");
   fprintf(fout,"%d %d %f\n",NUM_P,BLOCK_SIZE,time);

   CUT_SAFE_CALL(cutDeleteTimer(timer));


   for (i=0; i<nBodies; i++) {
      parts[i].x=bodies->p[i].x;
      parts[i].y=bodies->p[i].y;
      parts[i].z=bodies->p[i].z;
      parts[i].vx=bodies->p[i].vx;
      parts[i].vy=bodies->p[i].vy;
      parts[i].vz=bodies->p[i].vz;
      parts[i].mass=bodies->p[i].mass;
   }


   // clean up memory
   CUDA_SAFE_CALL(cudaFree(d_Planets));
}

float energy(int nbodies, struct planet * bodies)
{
   float e;
   int i, j;

   e=0.0;
   for (i=0; i<nbodies; i++) {
      struct planet * b = &(bodies[i]);
      e += 0.5 * b->mass * (b->vx * b->vx + b->vy * b->vy
               + b->vz * b->vz);
      for (j=i+1; j<nbodies; j++) {
         struct planet * b2 = &(bodies[j]);
         float dx = b->x - b2->x;
         float dy = b->y - b2->y;
         float dz = b->z - b2->z;
         float distance = sqrt(dx*dx + dy*dy + dz*dz);
         e -= (b->mass * b2->mass) / distance;
      }
   }
   return e;
}

void offset_momentum(int nbodies, struct planet * bodies)
{
   float px=0.0, py=0.0, pz=0.0;
   int i;
   for (i=0; i<nbodies; i++) {
      px += bodies[i].vx * bodies[i].mass;
      py += bodies[i].vy * bodies[i].mass;
      pz += bodies[i].vz * bodies[i].mass;
   }
   bodies[0].vx = -px / solar_mass;
   bodies[0].vy = -py / solar_mass;
   bodies[0].vz = -pz / solar_mass;
}

void create_particles(int len, struct planet * parts)
{
   int i;
   parts[0].x=0;
   parts[0].y=0;
   parts[0].z=0;
   parts[0].vx=0;
   parts[0].vy=0;
   parts[0].vz=0;
   parts[0].mass=solar_mass;
   for (i=1; i<len; i++) {
      parts[i].x=i;
      parts[i].y=0;
      parts[i].z=0;
      parts[i].vx=0;
      parts[i].vy=1/sqrt((float)i);
      parts[i].vz=0;
      parts[i].mass=1e-11;
   }
}


////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int
main(int argc, char** argv)
{

   printf ("Starting...\n");
   int n = 628;

   struct planet parts[NUM_P];
   create_particles(NUM_P, parts);
   offset_momentum(NUM_P, parts);

   doAdvance(n,NUM_P,parts,0.01);

   CUT_EXIT(argc, argv);
   return 0;
}
```

A.3 n-body-triangle_kernel.cu

Code showing the Triangle version (described in Section 3.3.2).

```
/*
 * n-body-triangle_kernel.cu
 *
 *  Kernel definition for the Triangle version.
 *
 */
#define NUM_BLOCKS 1
#define BLOCK_SIZE 512
#define STEPS 628
#define NUM_P 512

#define pi 3.141592653589793
```

```
#define solar_mass 1
#define days_per_year 365.24

struct planet {
   float x,y,z;
   float vx,vy,vz;
   float mass;
};
struct pass {
   struct planet p[NUM_P];
};


__global__ void
kernel_advance(int nBodies, struct pass * b, float dt, int steps)
{
   __shared__ struct pass local;
   int k;
   local=*b;
   for (k=1; k<=steps; k++) {

      ///// original advance()
      int i,j;
      for (i=threadIdx.x; i<nBodies; i+=BLOCK_SIZE) {
         for (j=i+1; j<nBodies; j++) {
            float dx = local.p[i].x - local.p[j].x;
            float dy = local.p[i].y - local.p[j].y;
            float dz = local.p[i].z - local.p[j].z;
            float distance = sqrtf(dx*dx + dy*dy + dz*dz);
            float mag = __fdividef(dt,
                  (distance * distance * distance));
            local.p[i].vx -= dx * local.p[j].mass * mag;
            local.p[i].vy -= dy * local.p[j].mass * mag;
            local.p[i].vz -= dz * local.p[j].mass * mag;
            local.p[j].vx += dx * local.p[i].mass * mag;
            local.p[j].vy += dy * local.p[i].mass * mag;
            local.p[j].vz += dz * local.p[i].mass * mag;

         }
      }
      __syncthreads();
      for (i=threadIdx.x; i<nBodies; i+=BLOCK_SIZE) {
         local.p[i].x += dt* local.p[i].vx;
         local.p[i].y += dt* local.p[i].vy;
         local.p[i].z += dt* local.p[i].vz;
      }
      __syncthreads();
      ///// end original advance()

   }
   *b=local;
}
```

A.4 n-body-square-small_kernel.cu

Code showing the Square version (described in Section 3.3.3).

```
/*
 * n-body-square-small_kernel.cu
 *
 *   Kernel definition for the Square version.
 *
 */


#define BLOCK_SIZE 256
#define NUM_P 512

#define pi 3.141592653589793
#define solar_mass 1
#define days_per_year 365.24

struct planet {
   float x,y,z;
   float vx,vy,vz;
   float mass;
};
struct pass {
   struct planet p[NUM_P];
};


__global__ void
kernel_advance(int nBodies, struct pass * b, float dt, int steps)
{
   __shared__ struct pass local;
   int k;
   for(k=0; k<nBodies; k++) {
      local.p[k]=b->p[k];
   }
   for (k=1; k<=steps; k++) {

      ///// original advance()
      int i,j;
      for (i=threadIdx.x; i<nBodies; i+=BLOCK_SIZE) {
         for (j=0; j<nBodies; j++) {
            float dx = local.p[i].x - local.p[j].x;
            float dy = local.p[i].y - local.p[j].y;
            float dz = local.p[i].z - local.p[j].z;
            float distance = sqrtf(dx*dx + dy*dy + dz*dz);
            float mag = local.p[j].mass*
                 __fdividef(dt,
                 (distance * distance * distance));
            if (i!=j) {
               local.p[i].vx -= dx * mag;
               local.p[i].vy -= dy * mag;
               local.p[i].vz -= dz * mag;
            }
         }
      }
      for (i=threadIdx.x; i<nBodies; i+=BLOCK_SIZE) {
         local.p[i].x += dt* local.p[i].vx;
         local.p[i].y += dt* local.p[i].vy;
         local.p[i].z += dt* local.p[i].vz;
      }
      __syncthreads();
      ///// end original advance()

   }
   for(k=0; k<nBodies; k++) {
      b->p[k]=local.p[k];
   }
}
```

A.5 n-body-square-large_kernel.cu

Code showing the Large Square version (described in Section 3.3.4).

```
/*
 * n-body-square-large_kernel.cu
 *
 *  Kernel definition for the Large Square version.
 *
 */
#define NUM_BLOCKS 1
#define BLOCK_SIZE 256
#define STEPS 628
#define CHUNK_SIZE 256
#define NUM_P 256*100

#define pi 3.141592653589793
#define solar_mass 1
#define days_per_year 365.24


struct pass_pos {
   struct float4 p[NUM_P];
};
struct pass_vel {
   struct float4 v[NUM_P];
};
struct share_pos {
   struct float4 p[BLOCK_SIZE];
};
struct share_vel {
   struct float4 v[BLOCK_SIZE];
};
struct chunk_pos {
   struct float4 p[CHUNK_SIZE];
};


__global__ void
kernel_advance(int nBodies, struct pass_pos * g_pos,
      struct pass_vel * g_vel, float dt, int steps)
{
   int k;
   for (k=1; k<=steps; k++) {

      ///// original advance()
      int i,j,c;
      for (i=threadIdx.x; i<nBodies; i+=BLOCK_SIZE) {
         float4 pos=g_pos->p[i];
         float4 vel=g_vel->v[i];
         for (c=0; c<nBodies; c+=CHUNK_SIZE) {
            __shared__ struct chunk_pos local_p;
            local_p=*(((struct chunk_pos*)g_pos)+
               c/CHUNK_SIZE);
            for (j=0; j<CHUNK_SIZE; j++) {
               float dx = pos.x - local_p.p[j].x;
               float dy = pos.y - local_p.p[j].y;
               float dz = pos.z - local_p.p[j].z;
               float distance = sqrtf(dx*dx
                           + dy*dy
                           + dz*dz);
               float mag = local_p.p[j].w *
                    __fdividef(dt,
                    (distance * distance * distance));
               if (i!=j+c) {
                  vel.x -= dx * mag;
                  vel.y -= dy * mag;
                  vel.z -= dz * mag;
               }
            }
            __syncthreads();
         }
         g_vel->v[i]=vel;
      }
      __syncthreads();

      for (i=threadIdx.x; i<nBodies; i+=BLOCK_SIZE) {
            g_pos->p[i].x += dt* g_vel->v[i].x;
            g_pos->p[i].y += dt* g_vel->v[i].y;
            g_pos->p[i].z += dt* g_vel->v[i].z;
      }
      __syncthreads();
      ///// end original advance()

   }
}
```