High-performance particle simulation using CUDA

Author: Mikael Kalms
Supervisor: Robert Strzodka
Examiner: Ingemar Ragnemalm
Opponent: Purani Mouragunusamy

Department of Electrical Engineering, Linköping University

June 3, 2015

Abstract

Over the past 15 years, modern PC graphics cards (GPUs) have changed from being pure graphics accelerators into parallel computing platforms. Several new parallel programming languages have emerged, including NVIDIA's parallel programming language for GPUs (CUDA).

This report explores two related problems in parallel: How well-suited is CUDA for implementing algorithms that utilize non-trivial data structures? And, how does one develop a complex algorithm that uses a CUDA system efficiently?

A guide for how to implement complex algorithms in CUDA is presented. Simulation of a dense 2D particle system is chosen as the problem domain for algorithm optimization. Two algorithmic optimization strategies are presented which reduce the computational workload when simulating the particle system. The strategies can either be used independently, or combined for slightly improved results. Finally, the resulting implementations are benchmarked against a simpler implementation on a normal PC processor (CPU) as well as a simpler GPU-algorithm.

A simple GPU solution is shown to run at least 10 times faster than a simple CPU solution. An improved GPU solution can then yield another 10 times speed-up, while sacrificing some accuracy.

Acknowledgements

I would like to thank Ingemar Ragnemalm for being flexible and supportive throughout this 7-year endeavour and enabling me to finish this on short notice just before University regulations were about to change to my disadvantage.

Gernot Ziegler for piquing my interest in GPU computing and finding the right inroad to a position at the Max-Planck-Institut für Informatik.

Robert Strzodka for providing expertise and a paid position with a work environment full of skilled people during my stay at the MPI-Informatik.

Purani Mouragunusamy for performing an opposition on this work with short notice and providing a lot of constructive feedback on the report itself.

All figures in the Overview of CUDA chapter are from NVIDIA's CUDA Toolkit v7.0 documentation.

Contents

1 Introduction

This report describes an algorithm for simulating systems containing more than 100.000 particles which affect each other through gravitational pull. The algorithm is designed to be run on a modern PC graphics card (GPU). The implementation language used is CUDA.

The simulation techniques also support non-gravitational forces and external forces, but those are outside the scope of this article.

1.1 Particle simulation

Many physical phenomena can accurately be modelled as systems of interacting particles. Planetary systems and molecular structures are two examples. Computing the orbits of planets and evaluating the inter-particle forces in molecular structures can be done numerically, but the computational time rises quickly with the number of particles involved.

The remainder of this report will concern itself with a class of particle simulations that are commonly referred to as "N-body gravitational simulations": the simulation of a set of massive particles which exert gravitational forces upon each other.

1.2 N-body gravitational simulation

N-body gravitational simulations are commonly used within physics and astronomy to understand how matter behaves on a grand scale [5]. Why does the Earth, Moon and Sun move the way they do? What is the large-scale structure of the universe and how did it evolve? Etc.

The motion of two interacting particles can be computed analytically. There is no analytical solution for computing the motion of more than two interacting particles; discrete, numerical methods have to be applied.

Most discrete methods are based on a two-step process; first, evaluating forces between all particles, and second, integrating velocities and positions of all particles for a small interval of time. The algorithm section of the paper focuses on methods which reduce the total number of force evaluations required. Two different strategies are used.

1.2.1 Virtual Particles

Nearby clusters of particles can be grouped into heavier, virtual particles. These virtual particles provide a means to perform approximate interaction calculations between a single particle and a cluster of distant particles without having to perform as many interaction calculations.

1.2.2 Force Nodes

All particles, when taken together, create a gravitational field that extends through the entire space. Placing an evenly-spaced grid of sampling nodes

("force nodes") across space and sampling most aspects of the gravitational field at each force node, allows us to use information from the force nodes to approximate the gravitational field at each real particle. This, too, reduces the total number of interaction calculations required while introducing an error with a well-understood upper bound into the result.

2 Motivation

2.1 Background

Hardware-accelerated 3D graphics became available to the general public at a reasonable price in the middle of the 1990s. The raw computational power of a graphics card (GPU) outperformed that of a desktop PC CPU already in 1995, and overall GPU performance has been rising exponentially ever since.

Initial generations of consumer-grade GPUs were designed to do one thing only - draw triangles. These GPUs could be used for general-purpose compute operations although the computation needed to be reformulated into a series of triangle drawing operations. The computational cores in these GPUs also had such low precision that results were not usable for most computational tasks. These were the two main hindrances that kept general-purpose GPU computing a very niche field.

More recent GPUs have increased precision in the computational cores to full 32-bit IEEE float precision and beyond. NVIDIA's CUDA language and toolkit is the first practical offering to allow the general public to write general-purpose computational code that runs on GPUs.

2.2 Goals

This thesis project had two main goals:

Implement algorithms which utilize non-trivial datastructures in CUDA - CUDA presents itself as a C-style language, but there are some restrictions in the language. How well-suited is CUDA to write code that employs complex datastructures?

Evaluate feasibility of CUDA for general-purpose computations - CUDA offers a parallel computing architecture which has very high peak performance. It also has some significant limitations compared to a general-purpose CPU. How close does one get to the maximum theoretical performance in a real-world CUDA application?

We chose to implement an N-body simulation in 2D to test the language out. It would allow for easy comparison between CPU and GPU performance. Applying algorithmic optimizations would also introduce various advanced acceleration datastructures. Thereby we would get a reasonably complex algorithm, with advanced datastructures, which has real-world applications.

A reasonable performance target would be to simulate 100.000+ particles with at least 10 integration steps per second.

3 Previous work

The motion of two interacting particles can be computed analytically. There is no analytical solution for computing the motion of more than two interacting particles; discrete, numerical methods have to be applied.

3.1 Methods for N-body simulation

Most discrete methods are based on a two-step process:

First, evaluate the forces between all particles. Then, integrate the velocities and positions of all particles for a small interval of time. Repeat the above two steps to simulate the next small interval of time. Etc.

The accuracy of the above approach depends on the correctness of the force evaluation and the size of the time step used for integration. Larger time steps mean less accurate simulation results. Several schemes have been devised to speed up the calculations. These reduce the number of direct particle-particle interactions at the expense of accuracy.

Below follows a partial overview of methods.

3.1.1 Direct Particle-Particle interaction

Direct Particle-Particle interactions is the simplest way to perform an N-body simulation. It boils down to the following:

1. For each particle, accumulate all forces that other particles exhibit on it.
2. Use an integrator (Euler or Runge-Kutta for instance)[2] to update the positions and velocities of the particles accordingly.
3. Proceed to the next time step.

The accuracy of this method is directly tied to the length of the time step used during integration. Smaller time steps give more precise results. Also, dynamically shortening the time step for sets of particles which are close to each other can give a better performance/accuracy tradeoff.

Computing direct particle-particle interactions between N particles has time complexity $O(N^2)$.

3.1.2 Particle-Mesh interaction

Described here[3], a mesh - usually a regular grid - is introduced into the space where the particles reside. Direct interactions between pairs of particles are replaced with interactions between individual particles and nodes in the mesh. Changing the resolution of the mesh is the main means for affecting simulation accuracy and computational performance.

The gravitational forces from all particles combined form a potential energy field throughout the space. Sampling the potential energy field at each node in

the mesh allows the mesh to be used as an approximate description of how the gravitational field varies throughout the space.

Once the gravitational field potential is known in each mesh node, the field potential can be evaluated in an arbitrary location in space - locate neighboring mesh nodes, and interpolate their potentials to get the approximate potential at the desired location.

Summarized as an algorithm it becomes:

1. For each node in the mesh, accumulate the gravitational potential that each particle contributes to that location.
2. For each particle, interpolate the gravitational potential of surrounding mesh nodes, and compute the gravitational acceleration vector.
3. Use an integrator (Euler or Runge-Kutta for instance)[2] to update the positions and velocities of the particles accordingly.
4. Proceed to the next time step.

Computing particle-mesh interactions between N particles and M nodes has time complexity O(N*M). This becomes useful when M ¡ N.

This method works well for particles which are distant from each other. It is highly imprecise for particles near each other though.

3.1.3 Particle-Particle/Particle-Mesh (P3M) interaction

Also described here[3], the P3M method combines the precision of Particle-Particle interaction with some of the performance from the Particle-Mesh interaction.

The gravitational forces that apply to a particle can be split into short-range and long-range components. The short-range component applies for pairs of particles which are near each other (say, within 2 to 3 units in the mesh). These pairs of particles should have their effect evaluated using direct particle-particle interactions. Pairs of particles further apart should have their effect evaluated using particle-mesh interactions instead.

Combining Particle-Particle and Particle-Mesh interactions provides a blend of accuracy and performance from both methods.

3.1.4 Barnes-Hut algorithm

Presented here[4], the algorithm introduces further accuracy/performance improvements over the previous methods.

The basic idea is to overlay a quadtree (or other N-dimensional spatial tree structure) over the space, and create a virtual particle for each subsquare in the tree that represents the combined center of mass for all particles within that square. This creates a level-of-detail representation of the full set of particles.

Then, when applying gravitational forces to particles, virtual particles are used instead of real particles. The further away, the coarser LOD is used.

This algorithm ends up with a typical complexity on the order of O(n log n).

4 Overview of CUDA

CUDA is a general-purpose parallel computing platform. It allows programmers to write code that will execute on GPUs by writing code in a subset of C.

4.1 CPU vs GPU

Some classes of algorithms execute considerably quicker on GPUs than on CPUs. Comparing a GPU with a CPU in the same price range shows differences in three key areas; instruction execution, memory access, and parallelization.

4.1.1 Instruction execution

A CPU is designed to execute a low number of disparate instruction flows at high speed. A lot of transistors are devoted to handling complex control flows efficiently.

A GPU is designed to execute large numbers of similar instruction flows at high speed. Lots of transistors are spent on raw computation. Complex control flows take a many clock cycles to complete and cause stalls. GPUs typically attempt to hide such stalls by switching to another group of instruction flows in the meantime.

Theoretical computational power for a GPU is much higher than for a CPU these days (Figure 1).

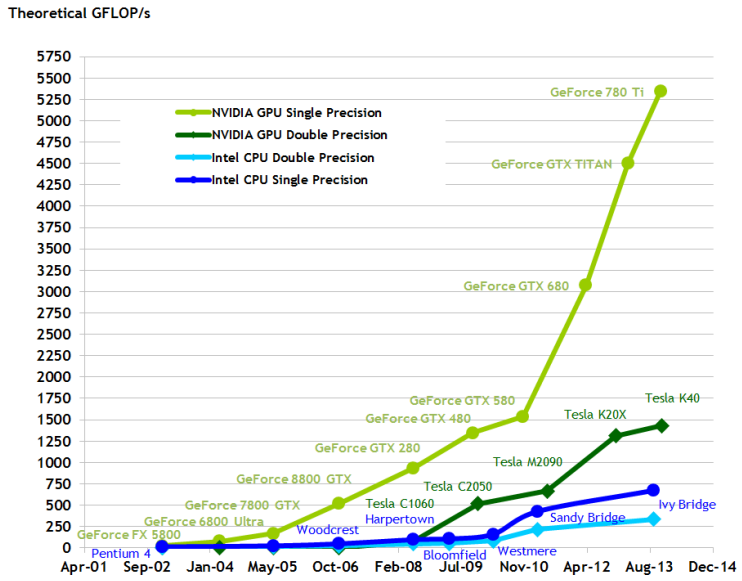

Figure 1: Theoretical CPU vs GPU floating point operational throughput [1]

4.1.2 Memory access

A CPU is designed to handle both sequential memory access and random memory walks fairly well. A lot of transistors are devoted to caching. Caches are designed to benefit most programs without requiring explicit cache control.

A GPU is designed to handle batch memory access quickly. There is typically some caching, but conscientious programs can conserve memory bandwidth and increase performance significantly by utilizing local memory stores that are shared by groups of computation units for intermediate processing.

Theoretical memory bandwidth for a GPU is much higher than for a CPU these days (Figure 2).

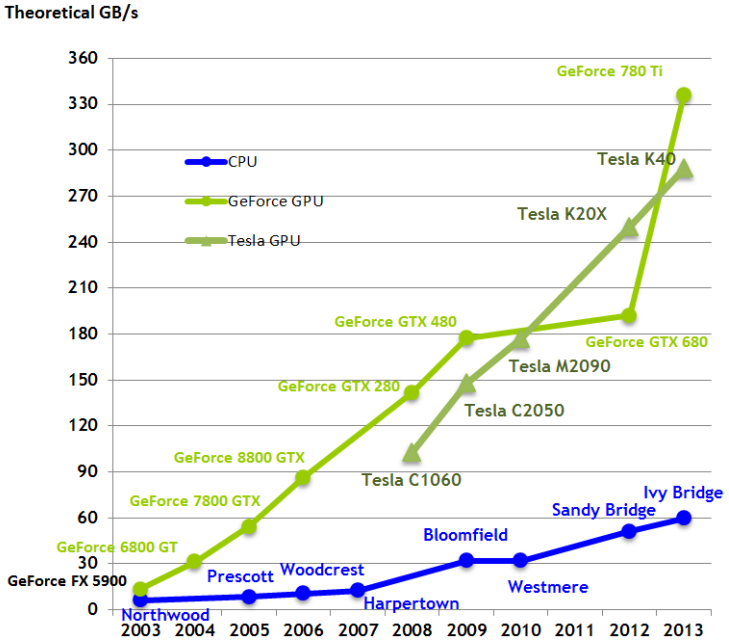

Figure 2: Theoretical CPU vs GPU memory throughput [1]

4.1.3 Parallelization

The basic programming model for CPU programs is non-parallel. A programmer can get fairly high performance out of a single-threaded CPU algorithm; moving to explicit parallelization is required to get full performance out of the chip. This typically involves running tens of threads in parallel. GPU programs, on the other hand, need to be written with a significant degree of parallelism present at a macro level for the GPU to be able to utilize more than 1

4.2 Programming model

This section will describe the programming model for CUDA 2.0. There are newer versions but the differences so far do not make a huge difference to the thesis project.

4.2.1 Kernels

A CUDA program is called a kernel. A kernel is a function that performs processing on a subset of a dataset. It is intended to be executed many times over, with different processing arguments, to process the entire dataset. Here is a kernel which will perform element-by-element addition of two vectors, if invoked correctly:

```
1
// Performs element-wise addition of two vectors: a = b + c
3
__global__ addVector(float* a, float* b, float* c)
5 {
  uint index = blockIdx.x * blockDim.x + threadIdx.x;
7 a[index] = b[index] + c[index];
}
```

The above kernel describes how to compute a single element in the output array. In order to compute all output elements, the kernel must be executed multiple times. CUDA organizes the multiple executions of the kernel as a single invocation which is comprised of a number of blocks and threads:

```
1 int main()
{
3 ...
  // Add two vectors which are of length N together
5 // Assuming here that N is a multiple of 16
  dim3 threadsPerBlock(16);
7 dim3 numBlocks(N / threadsPerBlock.x);
  addVector<<<numBlocks, threadsPerBlock>>>(A, B, C);
9 ...
}
```

The special threadIdx and blockIdx variables contain the ID of the current block within the grid, and the ID of the current thread within the block (see 4.2.2).

4.2.2 Blocks and threads

An invocation of a CUDA kernel can be represented as a grid. A grid is made up of a number of blocks. A block is made up of a number of threads (Figure 3).

A CUDA thread is similar in nature to a CPU thread; it represents one execution flow through the kernel. Threads within a block can coordinate between

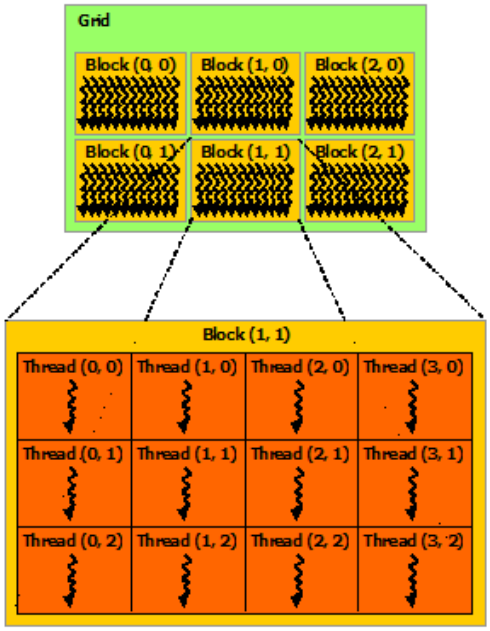

Figure 3: Grids, blocks and threads in CUDA [1]

each other. As an example, placing the __synchthreads() command within a kernel will make all threads within the block stop execution at that point and wait for all remaining threads to catch up before continuing execution through the kernel. Threads within a block can also communicate with each other via a small shared memory region. These synchronization and communication tools allow the threads within a block to collectively implement many complex algorithms.

A CUDA block is a group of threads. A block is CUDA's primary resource scheduling unit: first resources (registers, compute units, local memory) will be allocated for a block, then a block is executed, then the resources are returned. If there are enough hardware resources available then CUDA will process multiple blocks at the same time. Each block is scheduled independently of all other blocks. This means that a kernel must be written to give the same result regardless of which order the blocks are processed, and regardless of whether some blocks are processed in parallel.

4.2.3 Memory hierarchy

CUDA threads can access memory from three memory spaces (Figure 4).

Each thread can have some thread-local memory. It is intended for capturing temporary calculation results during execution of individual threads. This memory will lose its contents between thread executions.

Each block can have some per-block shared memory. This can be used for

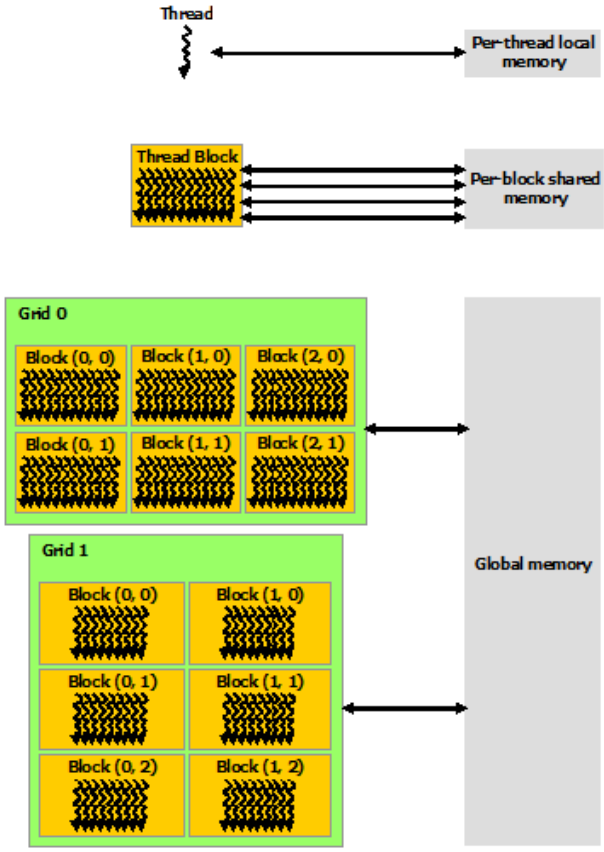

Figure 4: Memory hierarchy in CUDA [1]

sharing intermediate calculation results between threads in a block. It enables efficient coordination between threads. It can also be used as a manual cache, reducing usage of the global memory bus. This memory will lose its contents between block executions.

The global memory space is accessible by all grids, blocks and threads. This memory space includes all the main RAM on the GPU as well as the CPU's own memory. It retains its contents between kernel invocations.

4.3 Performance considerations

Here are some performance guidelines which apply when implementing most algorithms in CUDA.

4.3.1 Utilize all cores

CUDA-capable hardware is organized as a number of processing modules. Each module has a couple of computational cores and some local memory. Each block in a grid will be fully processed by one processing module. In order to utilize all cores, a kernel must be invoked as a grid with enough blocks to occupy all processing modules. Since processing time might differ between blocks, it is good practice to invoke the kernel as a grid with more blocks – say, 5-10 times as many – than the number of processing modules.

All CUDA-capable hardware so far also executes groups of 32 threads in parallel within a block. If the number of threads in a block is not an even multiple of 32, the modulo-32 remainder represents computational cores that will remain idle during processing of the corresponding 32-thread group.

4.3.2 Keep computational throughput high

Processing modules in CUDA-capable hardware organize threads in groups of 32 threads for scheduling purposes. These groups are called warps. All threads in a group are processed in parallel and in lock-step; that is, all threads in the warp will execute the same instruction at the same time. If flow control within the kernel code causes threads in the same warp to take different paths through the kernel, then the processing module will process each subset of threads separately, with a corresponding reduction in performance. Maximal computational throughput thus requires that all threads within a warp take the same execution path through a kernel. Threads within different warps can take different paths without affecting performance.

All threads in a warp will stall whenever one thread stalls due to memory accesses. Accesses to and from global memory are particularly slow. These stalls can be hidden by designing kernels such that one warp only needs a fraction of the available hardware resources in a processing module; a processing module will then manage multiple warps (and possibly multiple grids) in parallel, and the processing module will then process other warps while waiting for the first warp to be able to proceed.

4.3.3 Keep global memory bandwidth usage low

All CUDA-capable hardware to date has higher computational throughput than global memory bandwidth. Most algorithms will use all global memory bandwidth if implemented navely. Bandwidth consumption can be reduced by designing algorithms so that they read global memory linearly instead of as many small, scattered reads. Kernels which read or write data structures many times over can often be reorganized to copy a portion of the data structure from global memory into per-block shared memory or thread-local memory, then performing all operations against the non-global memory, and finally copying the modified portion of the data structure back to global memory before proceeding to the next portion.

4.3.4 Development time vs execution time

Designing algorithms to utilize CUDA-capable hardware well is a time consuming task. The development time spent results in a reduced execution time for the algorithm. There is a tradeoff to be made here; how much time is it worth to spend on development, given the expected execution time improvements? One thing is for certain: aiming for 100% efficiency is not a viable option. It is better to accept having imperfect resource utilization if it enables developing a correctly working and testable algorithm in a much shorter time frame.

5 Algorithms

5.1 Mathematical description of particles

A particle system is defined in this paper as a set of N particles $P_{1..N}$. Each particle P_i has a position p_i, a mass m_i and a current velocity v_i. The cumulative force acting on each particle is denoted F_i.

$$\frac{dp_i}{dt} = v_i \tag{1}$$

$$\frac{dv_i}{dt} = \frac{F_i}{m_i} \tag{2}$$

Each pair of particles attract each other with gravitational force, whose magnitude is dependent on distance d between the particles:

$$|F| = G\frac{m_i m_j}{d^2} \tag{3}$$

The force F_{ji} that particle P_j exerts over particle P_i and vice versa can be written as:

$$F_{ji} = -F_{ij} = G\frac{m_i m_j}{|p_i - p_j|^2}\frac{p_i - p_j}{|p_i - p_j|} \tag{4}$$

5.2 Two-step simulation

The simulation algorithm described in this paper has two main steps: evaluation of forces, and then integration of those.

5.2.1 Evaluation of forces

For each pair of particles, the force F_{ji} that particle P_j exerts on particle P_i can be described as:

$$F_{ji} = G\frac{m_i m_j (p_i - p_j)}{|p_i - p_j|^3} \tag{5}$$

Here, G is the gravitational constant.

The total force F_i exerted by all other particles on one particle P_i can be written as:

$$F_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} G\frac{m_i m_j (p_i - p_j)}{|p_i - p_j|^3} \tag{6}$$

The force evaluation step consists of evaluating F_i for all particles P_i.

Computing F_i directly for each particle has computational complexity $O(N^2)$. An algorithm which reduces the time complexity is detailed later in this chapter.

5.2.2 Integration of forces

The particles and the forces between them form a system of differential equations. The simplest explicit integration method is the Euler method.

Given a small, fixed time step Δt then the movement of a particle P_i can be computed as:

$$v_{i,new} = v_i + \frac{F_i}{m_i}\Delta t \tag{7}$$

$$p_{i,new} = p_i + v_{i,new}\Delta t \tag{8}$$

Performing Euler integration for all particles has computational complexity $O(N)$. This paper will not explore other integration algorithms.

5.3 Spatial organization

Since this is a 2D particle simulation, and particles are assumed to exist within the $(0, 0)$ - $(1, 1)$ portion of the coordinate system, a 2D grid can be overlaid over the coordinate system (Figure 5). Each particle can be assigned to one grid cell. Force evaluation is unaffected by spatial organization. Force integration may result in particles needing to move between the grid squares.

Figure 5: Particles organized into 2D grid

The set of particles can further be organized into a tree structure by recursively combining each 2x2 pair of grid cells into a higher-level grid cell. Figure 6 shows a five-layer tree structure where the top level incorporates all particles.

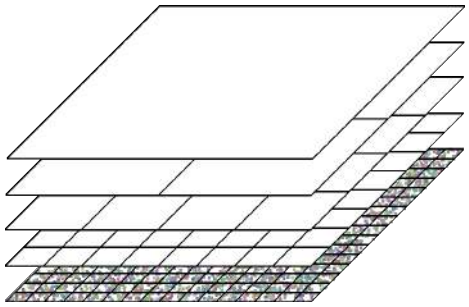

Figure 6: Particles organized into a hierarchical 2D grid

5.4 Ensuring particles can be parallel processed by CUDA

In order to make the computations fit well to running on a CUDA device, the algorithms presented here will employ two particular strategies:

- Particles will be managed in chunks of 64 particles at a time. This is enough to make the CUDA code run with full warp parallelism (needs at least a 32 particles).
- Particle chunks will be organized into a large number of blocks. This enables CUDA to distribute the processing across all the processing modules in the CUDA device.

5.5 Approximations and far particle-particle interactions

There are many strategies that can be employed to reduce the total number of particle-particle interactions as soon as the particles have been organized spatially. All those algorithms make trade-offs between accuracy and computation time.

Given that the force F between a pair of particles is related to the particle-particle distance d as $F \sim 1/d^2$, particles that are further apart impact each other less. This suggests that we should be able to use approximations with larger errors the further away particles are from each other while retaining a reasonable total error.

If we approximate the distance d between a pair of particles with $(1 \pm \epsilon_d) * d$ instead, where ϵ_d is a relative error bound less than 1.0, then the corresponding relative error bound in the force calculation $\epsilon_{F,d}$ will be:

$$\epsilon_{F,d} = \max |\frac{\frac{1}{(d*(1\pm\epsilon_d))^2}}{\frac{1}{d^2}}| - 1 = \max |\frac{1}{(1 \pm \epsilon_d)^2}| - 1 = \frac{1}{(1 - \epsilon_d)^2} - 1 \qquad (9)$$

This shows that with a distance error bound that increases linearly with the distance between particles (Figure 7), the relative error bound ϵ_{F_i} for the force computations for a particle will be directly related to the total number of particles in the system, regardless of how the particles are distributed in space.

The tree structure is useful for classifying particle interactions according to the above model: interactions between particles in the same or neighboring cells are classified as *near* interactions, whereas interactions between particles in non-neighboring cells are classified as *far* interactions. All near interactions need to be performed with full accuracy. However, far interactions can be made less accurate (Figure 8).

This scheme can also be applied hierarchically; out of all the cells containing particles suitable for far interactions, if interactions against the two innermost rows of cells are computed with a certain error, then the next four rows of cells can be computed with a twice as large absolute error, etc (Figure 9).

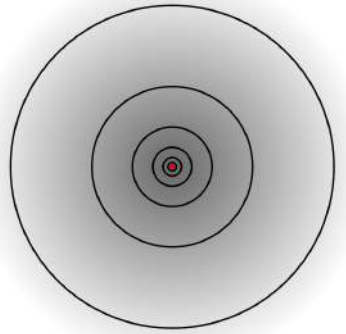

Figure 7: Max distance error increasing linearly
Each ring indicates where max distance error doubles

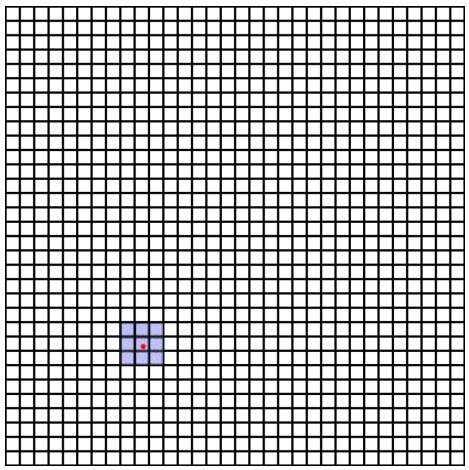

Figure 8: Near grid squares (blue) vs far grid squares (white)
Interactions for particles within near squares need to be performed with full accuracy

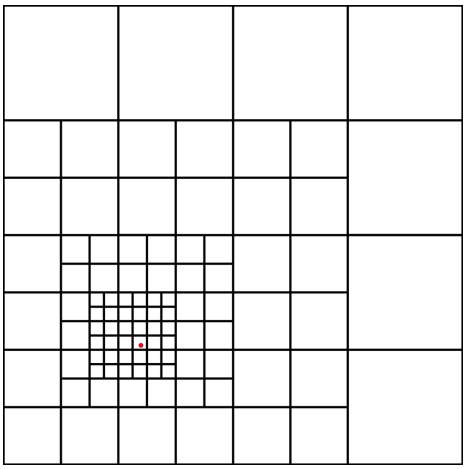

Figure 9: Tree structure determines distance-related error bound

5.6 Approximating far particle-particle interactions

The key factor that makes the particle simulation computation intensive is that evaluating forces between all particles has computational complexity $O(N^2)$. The following sections explore two strategies which reduce the computational complexity.

5.6.1 Virtual particles

One approximation we can do is to combine multiple particles together into virtual particles, and use these instead of the real particles when computing far particle-particle interactions.

We will define virtual particle creation as taking 2x2 grid cells, overlaying a regular 8x8 grid on top of those cells, sorting all the particles within the 2x2 cells into the regular 8x8 grid, and computing one virtual particle for each cell in the regular 8x8 grid. Each virtual particle will then be given the combined mass of all particles in the cell and be positioned at the gravitational center of those particles (Figure 10).

The virtual particles can be used instead of groups of real particles when computing far particle-particle interactions from a group of distant particles to a real particle.

5.6.2 Hierarchy of virtual particles

We can repeat the virtual particle creation multiple times. This provides us with a hierarchical representation of the virtual particles, where we have at most 64 particles per cell on each level. See Figure 11.

We can now devise a hierarchical scheme: when evaluating the particle-particle interactions for a group of particles within a single grid square, first

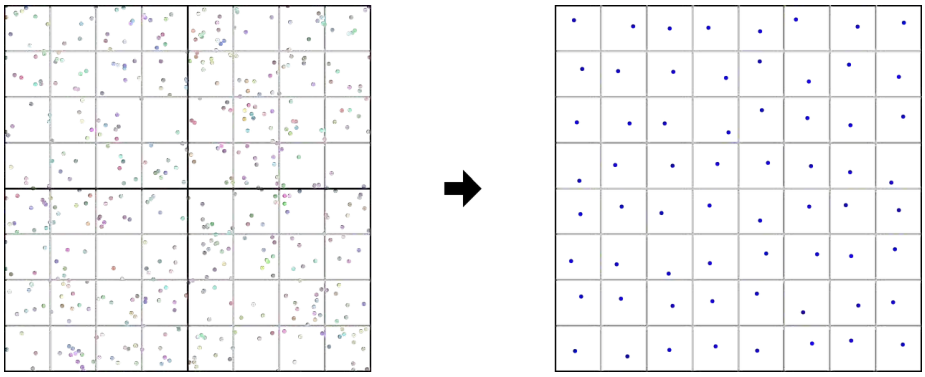

Figure 10: Creating 64 virtual particles from 2x2 grid cells

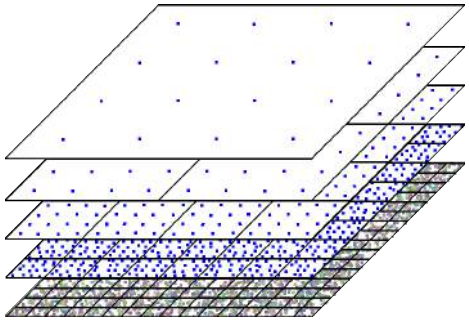

Figure 11: Particles with hierarchy of virtual particles

evaluate far particle-particle interactions from the topmost layer of virtual particles against the group of real particles; then, move down one level in the tree, and repeat the evaluation process from those grid squares which were considered near on the prior level but far on the current level. Repeat until the bottom-most level has been reached; handle remaining particle interactions as near particle-particle interactions. See Figure 12.

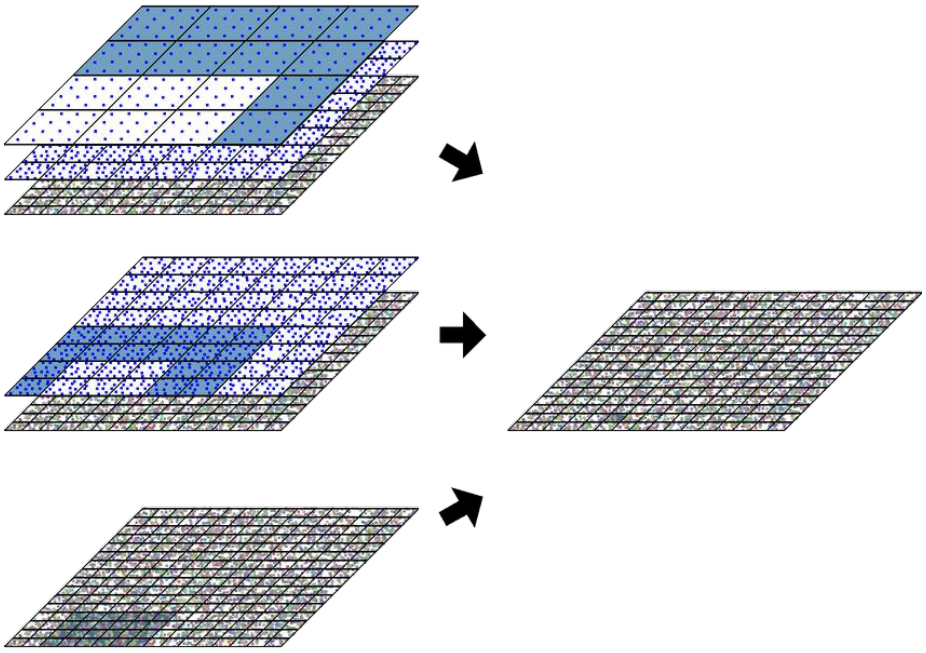

Figure 12: Virtual particles affecting real particles hierarchically

5.6.3 Evaluation algorithm using a hierarchy of virtual particles

In order to perform a full evaluation of particle-particle interactions using a hierarchy of virtual particles, we would go through the following steps:

1. Choose a maximum tree depth. This will correspond to the total number of tree levels L.
2. Starting with level L-1 and going up to level 2, take each 2x2 cell pair from level l and compute a set of 64 virtual particles for level $l-1$.
3. Starting at level 2 and going down to level $L-1$, Evaluate the new far particle-particle interaction between virtual particles in level l and particles at the bottommost level.
4. Go through the real particles in level $L-1$. Evaluate near particle-particle interactions.

This algorithm results in considerably less than N^2 force computations for a particle with N particles.

5.6.4 Force nodes

The number of far particle-particle interaction computations can be reduced further for dense particle systems by placing a layer of nodes evenly spaced throughout the coordinate system (Figure 13), and evaluating far particle-particle interactions for each node. We will call these nodes "force nodes". Force nodes are virtual particles whose position are predetermined and whose mass is 1.0.

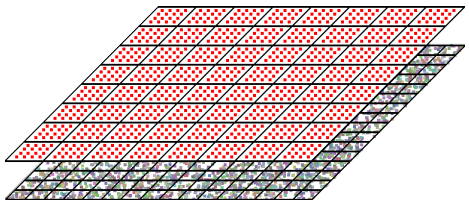

Figure 13: Particles with a single layer of force-nodes

Once we have evaluated the force at each node, we have effectively sampled the gravitational field at regular intervals, but specifically for far particle-particle interactions.

For each real particle, we find the surrounding 2x2 force nodes, interpolate the forces at those nodes linearly, and multiply with the real particle's mass. This gives us an approximation of the effect of far particle-particle interactions at the particle. We will complete the force calculations for that particle by manually computing all near particle-particle interactions and adding that up with the result from interpolation of the forces from the nearest force nodes.

5.6.5 Hierarchy of virtual particles and force nodes

We can consider the force nodes to exist hierarchically as well (Figure 14).

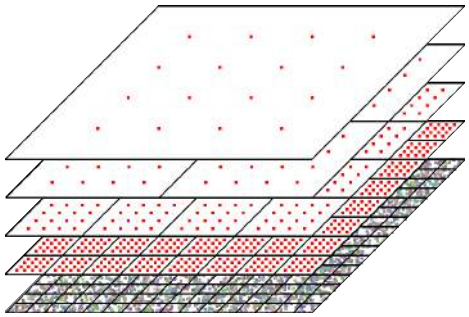

Figure 14: Particles with hierarchical layers of force-nodes

With both these structures in place we can devise another hierarchical scheme: when evaluating the far particle-particle interactions for a level of force nodes, first interpolate the forces from the level of force nodes on the level above in the hierarchy. This will provide results for the bulk of far particle-particle interactions. Combine those results with far particle-particle interactions for those regions that were classified as near particle-particle interactions on the level above, but are classified as far particle-particle interactions on the current level; that will yield the total far particle-particle interactions for force nodes on the current level. Finally, near particle-particle interactions will be performed between real particles on the bottom-most level to compute the close particle interactions.

This scheme results in the first two levels of the hierarchy not being used, and a direct mapping between virtual particle levels and force node levels (Figure 15).

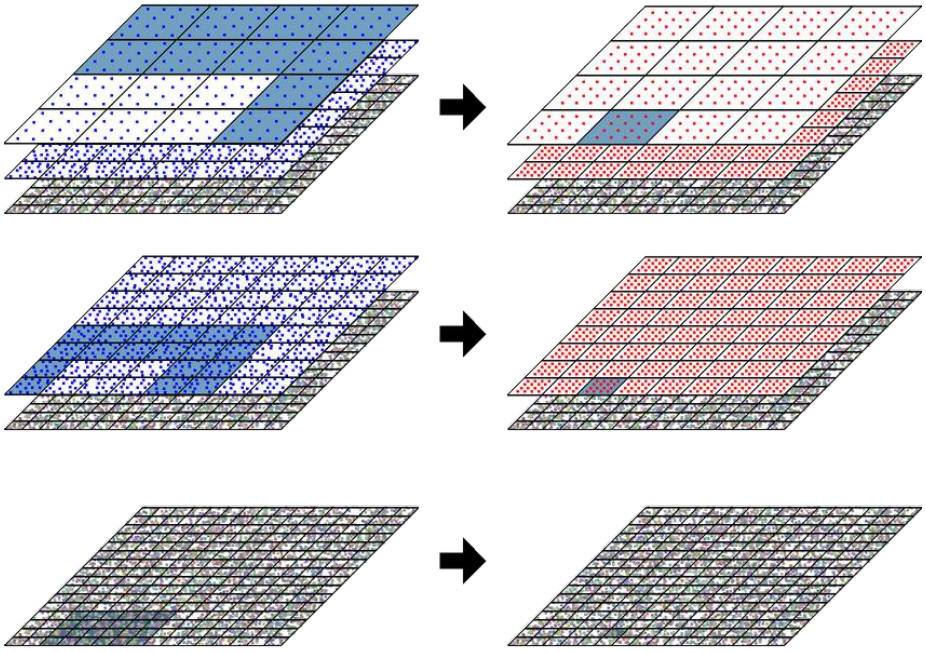

Figure 15: Virtual particles affecting force nodes hierarchically

5.6.6 Evaluation algorithm using a hierarchy of virtual particles and force nodes

In order to perform a full evaluation of particle-particle interactions using both virtual particles and force nodes, we would go through the following steps:

1. Choose a maximum tree depth. This will correspond to the total number

of tree levels L.

2. Starting with level L-1 and going up to level 2, take each 2x2 cell pair from level l and compute a set of 64 virtual particles for level $l-1$.

3. Starting at level 2 and going down to level $L-1$,

   (a) Evaluate the new far particle-particle interaction between virtual particles and force nodes in level l.

   (b) Interpolate the force values among the force nodes in level l and apply them to force nodes or real particles in level $l+1$.

4. Go through the real particles in level $L-1$. Evaluate near particle-particle interactions.

This introduces another tradeoff between accuracy and performance, compared to the hierarchy of virtual particles evaluation algorithm.

6 Implementation

6.1 Core datastructures

6.1.1 Particle

A particle is represented by a 2D position vector, a 2D velocity vector and a mass. Each particle is assigned a unique ID; this allows us to to identify particles even if the simulation code has shuffled them around within the data structures. The particle color is only used for visualization.

6.1.2 ParticleChunk

Particles are never managed one at a time. The basic container for particles is the ParticleChunk. It is a fixed-size container with room for 64 particles. Having a basic unit of up-to-64 particles enables parallelization on a micro level in all computations. When working with more than 64 particles, several ParticleChunks will form a linked list. One linked list of ParticleChunks is a suitable work unit for one CUDA block.

6.1.3 ParticleGridLevel

Particles are sorted into a regular-spaced 2D grid. This is the ParticleGridLevel. Each square in the grid contains one linked list of ParticleChunks. Each grid square will be processed as a separate block by CUDA. If there are enough many blocks then an entire GPU can be utilized to perform particle computations.

Sorting particles into chunks relies on particles being spatially distributed. If all particles happen to be very close, processing will degenerate to utilizing only a small subset of the hardware's capabilities. This paper does not attempt to construct a solution that handles those cases with retained high performance.

The ParticleGridLevel can be thought of as a rough spatial organization of the particles. It enables several higher-level optimization strategies, as will be shown in the following sections.

6.1.4 ParticleGrid

A ParticleGrid is a set of ParticleGridLevels with increasingly coarse-grained grid squares. The ParticleGridLevel at level N consists of 2^N x 2^N squares; with the top-most ParticleGridLevel being just a single square. The ParticleGrid itself represents a hierarchial partitioning of the particles. It is a building block for acceleration algorithms.

6.2 Programming strategies

6.2.1 One kernel invocation per tree level

CUDA is not well suited for processing tree-based datastructures. Grids apply directly to CUDAs processing model though. In order to process a tree where

each level is a grid, perform one kernel invocation per level in the tree. The invocations for the top few levels of the tree datastructure will not be able to utilize all the computational resources of the CUDA device, but that is not a big problem – the bulk of processing will be in the bottommost levels.

6.2.2 Manipulate datastructures in CUDA code, not CPU

Do not attempt to mix CUDA and CPU logic. It is easier in the long run if CUDA-side code has all creation, modification and destruction of all datastructures. Develop serialization/deserialization routines which run on the CUDA side that import/export data to a format that is easy for the CPU-side code to work with.

6.2.3 Effective layout of a CUDA kernel

Most kernels should be structured as follows:

- Load a part of the data structure into shared memory
- Process part of the data structure
- Write part of the data structure back to global memory

Shuffling the most commonly processed (and potentially to-be-modified) data structure into shared memory before doing any processing ensures that the data structure is read (and possibly written) only once. This ensures that the kernel uses the memory bus to global memory as efficiently as possible. Otherwise, access to global memory will often be the primary performance bottleneck when executing the kernel.

6.3 Writing robust CUDA code

When implementing algorithms in CUDA it is easy to spend lots of time tracking obscure bugs and squeezing out the last 5% of performance. Here are some guidelines for how to balance development time, execution time and implementation robustness against each other.

6.3.1 Implement a serial CPU version

Make an initial implementation as normal CPU code. Make it as simple as possible. This implementation will be the reference implementation which is is used to gauge the correctness of any subsequent implementations.

6.3.2 Implement a serial CUDA version

Make a 1-block, 1-thread implementation of the algorithm in CUDA. This is a good time to design CUDA-friendly data structures.

6.3.3 Implement a parallelized CUDA version

Make the algorithm distributed across multiple threads and multiple blocks. Ensure that there are many more blocks than there are processing modules in a typical CUDA device. Ensure that there are at least 32 threads per block. Ensure that processing one block only requires a fraction of the resources of one processing module on a typical CUDA device, so that each processing module can have several blocks in-flight at the same time.

6.3.4 Implement a CUDA version that uses shared memory

Make the algorithm use shared memory to minimize the amount of global memory access.

6.3.5 Test implementations against each other

Keep all implementations alive throughout development. Use them as tests for each other; if three implementations give the same result while the fourth produces a different answer, there is probably something wrong in the fourth implementation.

6.3.6 Run both in normal mode and device emulation mode

CUDA offers a "device emulation mode". This is a mode in which the CUDA code is run through a CPU-side emulator. Its primary purpose is to enable debugging with a CPU debugger. It can also serve as a slow CUDA device with very different characteristics than hardware-based CUDA devices; threads within a warp are not strictly processed in atomic parallel fashion when under emulation, the emulator does not process multiple blocks in parallel, threads and block execution will be scheduled differently than on real hardware, etc.

All these differences mean that latent bugs that will not show up on a real hardware-based CUDA device may show up when the same code is run under the emulator. Therefore, running all the implementations both in normal mode and in device emulation mode gives even broader test coverage.

6.3.7 Treat CUDA datastructures as opaque objects on CPU side

There is no difference in type between a pointer to CPU memory and a pointer to CUDA device memory. This makes the compiler unable to catch when pointers are being passed between CPU and CUDA code without conversion. Introduce a template class, CudaPointer⟨T⟩ or CudaObject⟨T⟩ or similar and use that when managing CUDA-side pointers in the CPU-side code; this provides type safety and ensures that CPU-side code will never accidentally access CUDA-side data structures and vice versa.

7 Results

7.1 Test hardware

The test results have been captured on a machine with the following specification: Intel Xeon E5-1650 V3 @ 3.5GHz Windows 8.1 32GB RAM NVIDIA GeForce GTX 980

7.2 Performance

The benchmark CPU implementation is a naive single-threaded implementation. The following steps could be done to improve performance: - Multithreading would provide a 12x speedup under ideal conditions - Using SIMD instructions would provide a 2x speedup (estimated) All in all, roughly a 25x speedup should be possible for the CPU implementation with some extra work.

Figure 16 shows a comparison of the performance of four algorithms:

- CPU (Optimized) assuming 25x speedup compared to the naive algorithm
- GPU (Brute-force)
- GPU (via Force Nodes) using a hierarchy of both virtual particles and force nodes
- GPU (Virtual Particles) using only a hierarchy of virtual particles

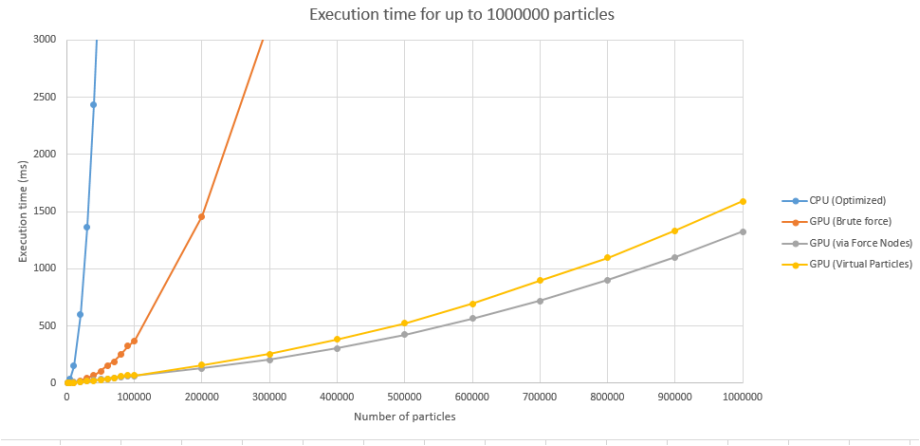

Figure 16: Performance results

Even at 100.000 particles, the GPU brute-force implementation is 40x faster than an optimized CPU implementation would be. In addition, the algorithms described in the paper are roughly 5x faster than the GPU brute-force implementation.

The graph shows that the effective time complexity for both algorithms described is somewhere between $O(N)$ and $O(NlogN)$. The time complexity is kept down by choosing different tree depths depending on the average density of the particle system; a tree depth of 5 was usually best for up to 100000 particles, while a tree depth of 6 was preferable for 100000-1000000 particles.

7.3 Accuracy

The results in Figure 17 show that the force vectors computed by both methods have roughly the same direction. However, Figure 18 shows how the average length of the calculate force vector is off by more, the more particles are computed. This is indicative not of a problem with the algorithm itself, but rather of precision problems: all computations in reference and CUDA implementations are performed with single-precision floating point, and no care is taken to avoid loss of precision when accumulating the intermediate results.

A more exact result - and more exact error bound - could be established by performing the calculations at higher precision, or by structuring the accumulation differently. This is not explored in this paper.

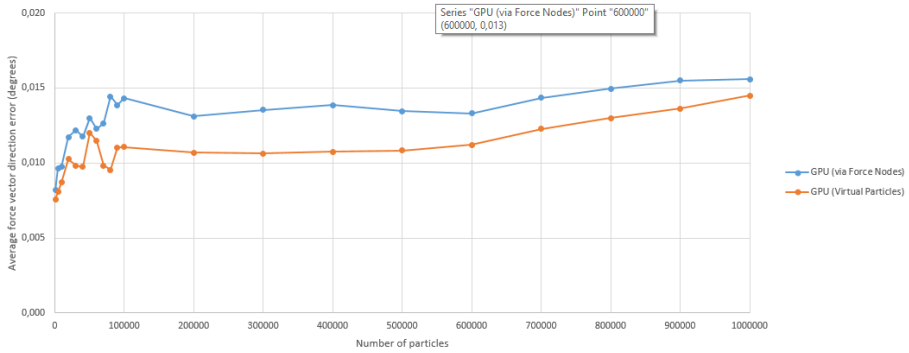

Figure 17: Average error in force direction, in degrees

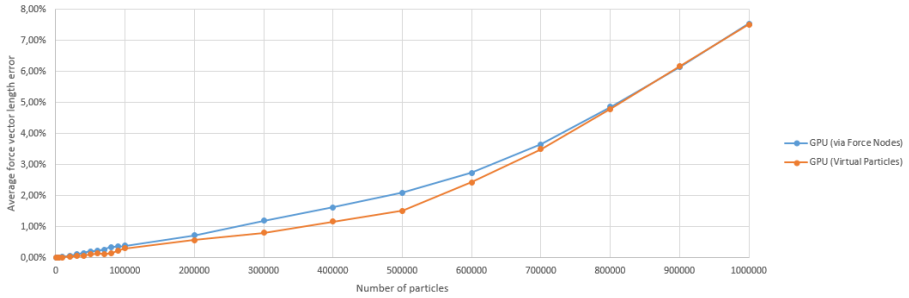

Figure 18: Average error in force vector length

8 Future work

There are several ways in which this work could be expanded. Here are some ideas.

8.1 Improve accuracy

Perform all accumulations in double precision? Use an approach when accumulating force vectors that is known to be robust against loss of precision during accumulation chains?

8.2 Expand into 3D

Changing from 2D to 3D will introduce several complexities; the error bound will result in more near particle-particle interactions. Also, the virtual particle creation will need to be rewritten to be as precise without using much more shared memory.

8.3 Support variable mass

This paper has assumed that all particles have equal mass. What does it take to keep errors bounded if particles have wildly varying mass?

8.4 Adaptive subdivision

The current algorithm will not utilize all GPU resources effectively if many particles gather within a small area of space. An adaptive subdivision mechanism would help in such a case. How does one implement that in CUDA effectively?

8.5 Allow particles to move outside the (0,0)-(1,1) space

The current algorithm requires all particles to be within a constrained space. Would an adaptive subdivision model allow for efficient simulation of systems where particles are not spatially constrained?

9 Summary

Key takeaways:

- Hierarchical algorithms can allow simulating interacting particle systems at between $O(N)$ and $O(NlogN)$ time complexity
- Implementing algorithms efficiently in CUDA requires the algorithms and the data structures to be designed in tandem
- Making a CUDA algorithm bug-free requires a strong automated regression test suite

References

[1] *Cuda Toolkit v7.0 documentation*. NVIDIA Corporation. URL: `http://docs.nvidia.com/cuda/index.html`.

[2] Alejandro Garcia. *Numerical methods for Physics*. Prentice-Hall, 1994.

[3] R.W. Hockney and J.W. Eastwood. *Computer Simulation Using Particles*. CRC Press, 1985.

[4] J.Barnes and P. Hut. "A Hierarchical O(n log n) force calculation algorithm". In: *Nature* v. 324 (1986).

[5] *N-Body Simulation*. Wikipedia. URL: `http://en.wikipedia.org/wiki/N-body_simulation`.