Processing Hard Sphere Collisions on a GPU Using OpenCL

Zachary Langbert[1] **and Mark C. Lewis**[1]
[1]Computer Science, Trinity University, San Antonio, TX, USA

Abstract— *Physically accurate hard sphere collisions are inherently sequential as the order in which collisions occur can have a significant impact on the resulting system. This makes processing hard sphere collisions on parallel hardware challenging. We present an approach to solving this problem that can be implemented using OpenCL that runs on current hardware. This approach makes significant use of atomic operations to prevent race conditions, even across thread groups. We find that an unoptimized implementation of the approach provides speed on modest GPUs that is on par with our earlier OpenMP parallel CPU approach and the OpenCL running on a CPU is faster than the OpenMP code. Full timing results using commodity GPU and using OpenCL on multi-core chips are presented.*

Keywords: Simulation, collisions, parallel, discrete-event, GPU

1. Introduction and Related Work

Many problems in the field of simulation involve collisions between bodies/particles. In this paper we will focus on particles that are represented as spheres. These types of collisions are typically modelled in one of two ways. Soft sphere collisions allow the bodies to overlap and restoring forces are applied at intervals to cause them to bounce. Hard sphere collisions treat the collisions as instantaneous events where an impulse is applied to the particles to produce the bounce.

Collision detection has been done on GPUs for a long time. For example, Kolb et al. used pixel-shaders before tools like OpenCL and CUDA were available and used depth maps stored in texture memory to make the runtime more efficient [4]. Soft sphere collisions can also be implemented efficiently using general N-body methods [1], [2]. The soft-sphere approaches are also applicable to other problems that involve interactions between nearby particles such as SPH codes [12]. The processing of soft-sphere collisions is basically a problem of detecting proximities between bodies, as the exact time of overlap is not resolved or dealt with. Separate GPU implementations exist for solving this more general problem for both simple and complex geometries [5].

Physically accurate hard-sphere collisions are more challenging. They are basically discrete event simulations, and the time ordering of the events is important. Any given collision can alter the ones that follow it. Many collisions can be done in parallel, assuming that they are far enough apart spatially so that the result of one doesn't alter the other. We have previously created methods for doing this on multi-core machines [7], [11], [6]. GPUs present a number of different challenges, and the previous methods will not work well in that context. Not only does efficient use of the GPU require that more threads be active at any given time, workloads on GPUs are best split across multiple thread groups and synchronization across the thread groups is more challenging than inside of the thread groups. For this reason, the use of a single shared queue structure becomes ineffective.

There are also multiple steps to the collision finding process. Some of them involve building data structures that are used for fast searching of the particles for collisions. These steps could be done on the CPU then copied to the GPU, but that would significantly degrade performance. Ideally, we want all the work to be done on the GPU so that the only data that is moved back to the CPU is particle data, and that should only be done when required for I/O.

Playne and Haywick present work on doing hard-sphere collisions using a multi-GPU approach [3], [13]. Their work included both soft-sphere elements with particle-particle forces and hard-sphere interactions when the particles get sufficiently close. However, their approach to hard-sphere interactions use posteriori collisions instead of discrete event priori methods. This means that the particles are allowed to advance to the end of the time step, then they are checked to see if they overlap at the end and corrections are applied to handle the collisions. These methods are not as accurate as using the priori discrete-event approach described here, they can miss collisions if the time step is too long, and they don't do a good job of resolving multiple collisions in a single time step in dense systems. Posteriori methods are easier to process though, especially on GPUs.

Our goal in this work is to deal with the problem of creating a physically accurate, discrete-event, priori algorithm for doing hard sphere collisions that can allow long time steps and run efficiently on a GPU. Section 2 outlines an algorithm for doing this in a basic multithreaded environment. Section 3 describes in detail the algorithm we have developed for doing this on a GPU. This is followed by sections showing the results of a basic implementation of this algorithm and our conclusions.

2. Multicore, Threaded Algorithm

To facilitate the discussion of the GPU approach, it is advantageous to begin by looking at a rough outline of the approach taken in a single-threaded implementation

and how that is modified for multiple threads on a multi-core processor. The single-threaded implementation can be described by the following pseudo-code [9].

1) Build spatial data structure.
2) Find potential collisions based on initial conditions and add them to a priority queue.
3) While there are potential collisions on the priority queue.
    a) Remove the next item from the priority queue.
    b) Process that collision.
    c) Remove all future potential collisions involving either of the colliding particles from the priority queue.
    d) Find new potential collisions involving those particles based on their new trajectories. Add those happening in the current time step to the priority queue.

This algorithm uses the term "potential collision" to refer to a triple of two particles and a time, where those two particles would impact at that time given their current trajectories. The word "potential" is significant because many of these won't actually come to pass if an earlier collision alters the path of either of the particles involved. Step 3c can remove many potential collisions when one is processed. Unfortunately, there is no simple way to know if a potential collision will actually be processed as a real collision without running through and processing them to find out. It is worth noting that because of this step, the standard priority queue implementation of a binary heap is not efficient.

This algorithm can be updated to work on multiple threads with a few alterations. The approach to parallelizing the building of the spatial data structure varies by data structure. Some care must be take to avoid race conditions when particles are added in. It is also possible to parallelize the discovery of the initial collisions as long as the priority queue is thread-safe or is locked on each add operation.

The parallelization of the processing loop is more interesting and requires a bit more information about the nature of collisions. Of primary significance is that while the order of collisions is important, information about collisions is propagated at a finite speed. That means that two collisions that are sufficiently far apart can be processed in parallel assuming that they happen close enough together in time. It simplifies things to use the simulation time step as a conservative estimate of the time. The spatial data structure can be used to keep track of where collisions are being processed at any given time to determine if the next one on the queue is safe or not. Here again, the priority queue needs to be thread safe or we must do locking to prevent multiple threads from altering it at the same time.

This approach has been shown to work well for tens of threads, but it doesn't scale well for a GPU where we would ideally like to have thousands of threads. In that situation, the single priority queue becomes a bottleneck that will not scale efficiently.

3. GPU Algorithm

The general outline of the algorithm shown above is maintained when we move to the GPU. First, we need to build our spatial data structure, then we find the initial potential collisions, then we run through and process the collisions. We will look at how each of these steps is adjusted to the GPU here.

3.1 Kernel 1 - Building the Spatial Data Structure

To keep things simpler for this project, we use a regular 2-D grid as the spatial data structure. Each grid cell keeps a list of the particles whose centers are located in it. This can be done with two arrays of integers. One is the size of the grid and stores the index of the first particle in that cell. The second has the same length as the number of particles and stores the next particle found in the grid cell. All elements of both arrays begin with a value of -1 to denote no link.

This process is done with threads spawned on a per particle basis. This opens the possibility of race conditions on the grid values storing the head as every particle in a given cell could, in the worst case, be processed at the same time. Fortunately, the order of particles doesn't matter and OpenCL supports atomic operations on any primitive value [14]. The operation of adding to the front of one of these lists can be done with an atomic read/set. This is done on the value in the grid that stores the first particle currently in that cell. It returns the previous value, which is stored in the second array at the location of the particle that is the new head.

The fact that only one thread will be responsible for a given particle means that we never have a situation where two threads alter the same location in the second array. Every atomic read/set on the grid will find a different value, so the linked list will be correctly built regardless of when the links in the second array are stored.

The grid based approach has been used previously both for sequential and parallel code [9], [10], [7]. The key is that the grid cells need to be large enough that particles in one cell can only collide with particles in adjacent cells during a given time step. That is to say that

$$\Delta x = r_{max} + c \times \Delta t \tag{1}$$

where Δx is the size of the grid cell, r_{max} is the largest particle radius in the simulations, Δt is the length of the time step, and c is a value several times larger than the velocity dispersion. This approach works well when the system includes forces other than those modelled with discrete events. It is also possible to pick a more arbitrary grid size and model the passing of particles from one grid cell to another as events, but that is not the approach we take here.

3.2 Kernel 2 - Finding Initial Potential Collisions

After we have built the spatial data structure that tells us where all the particles are located, we can use it to find the initial potential collisions. This work is done with threads allocated per cell. Instead of having a single queue of all the potential collisions, we keep one queue per cell. The fact that there should be few particles per cell makes it feasible to use something as simple as a sorted array for the queue. This could be changed to a binary heap if the queues were longer, but that introduces some overhead and because of the way that work is distributed, it is probably better to use a finer grid to keep the queues shorter than to improve efficiency for larger queues. This will be an area of future study.

To find the potential collisions, we have each cell check the particles in it against the others in that cell as well as all particles in the cells that are in the cell to the right and the three cells below it. Checking against more cells would cause potential collisions to be double counted. Any pairs that are found to collide are stored, along with the potential collision time, in the queue for that cell.

This can be done with a 3-D array where the third dimension is the potential collisions, but this approach is inflexible and can lead to a lot of wasted memory as we have to make the third dimension large enough to handle whatever collisions might occur in a single cell. An alternative approach is to again use a pool of memory and have the queues stored as linked lists in that pool. We keep a single value for the number of potential collisions currently stored and this variable is altered using an atomic read/increment operation. This tells us the next pool element to store a potential collision in and increments it so that no two threads for the cells will write to the same location in the pool. The potential collisions are first added to an array in local memory, then they are moved as a block to the global memory once the search is complete. This minimizes the number of atomic operations to one call per thread and groups the memory move to reduce the number of accesses to global memory, which is generally much slower than local memory on GPUs. Unfortunately, at this time there is no memory copy directive in OpenCL. Such a directive would have benefits to this segment of the code were it to be added in the future.

Using this approach, the memory overhead is greatly reduced as the total size of the pool needs to scale as the total number of potential collisions, not the maximum number in one cell. This means that non-uniform geometric distributions don't lead to significant wasted memory.

At the end of this process, we will have all of the initial potential collisions for each cell in unsorted lists.

3.3 Inside while Loop

Kernels 3-5 happen in a while loop that executes as long as there are more collisions left to process in the current time step. These steps are broken into separate kernels primarily for synchronization purposes. Each one must fully complete before the next one begins.

3.3.1 Kernel 3 - Sort Initial Potential Collisions

The third kernel is again done by allocating threads by cells in the grid. This pass only sorts the lists that were produced in the previous kernel. We have implemented this as a simple insertion sort. Here again, we expect the number of potential collisions in a given cell to be small. In an ideal configuration the average would be of order unity so the sorts should be doing very little work. To improve the memory performance, we first walk the list of potential collisions in each queue and copy it to a local array. Then we do the insertion sort on that array and copy the potential collisions back into the same locations in the list, preserving the earlier links. This approach is taken to reduce the number of accesses to global memory.

3.3.2 Kernel 4 - Processing Collisions

The fourth kernel does significantly more work. as this is where we actually process the potential collisions. As was discussed above, the primary challenge of hard-sphere collisions is that the order in which they are processed is significant and we can't process all the collisions simultaneously. The spatial data structure gives us a simple way to handle this.

Handle Safe Collisions

In this pass one thread is spawned per cell and each thread compares the time of its first collision to that of the adjacent cells. Only those threads which have a collision time lower than their neighbours will actually process a collision. Cells that have no potential collisions don't process and aren't considered in the comparison to see if a cell is the local minimum.

There is no synchronization needed for this, because the times on the first collisions are only read, not written to, and one particle can only take place in collisions that are in adjacent cells. Given that no two adjacent cells can be processing at the same time, it is safe for each thread to write out and change the particle positions and velocities when a collision is processed.

Mark Indices in Collided Particles Array

In addition to processing the collisions, each thread that does process a collision does one additional task. It stores an appropriate value in an array of the same length as the number of particles that we call the "collided particles array". The purpose of this array is to keep track of which particles were involved in a collision in that pass through the grid. This information is used in the next kernel.

There are no synchronization issues with this task for two reasons. First, given the way that we determine if a collision can be processed, no particle could be involved in two collisions at one time in this pass. What is more, even

if two threads could process the same particle, they would write the same value out to the array so it wouldn't matter which write happens first, and no data is read from this array in this kernel.

3.3.3 Kernel 5 - Delete Marked Potential Collisions and Find New Potential Collisions

The last kernel cleans up the potential collisions lists for each grid cell, then finds new collisions based on the particles in each cell that had just been involved in collisions. The first half of this work involves running through the list of potential collisions in each cell and checking each potential collision to see if one of the particles in it was collided during this pass. This is the first use of the collided particles array that was initialized in the previous kernel. In this kernel, that array is read only. The loop runs through the linked list of potential collisions, and if either the first or second particle involved has been marked in the collided particles array, that node is marked as unused and linked around so that it is no longer considered to be a potential collision.

After that has been done, each thread then goes looking for new collisions involving any of the particles in that cell that had been in a collision. This is done by walking the list of particles in the cell, and checking if that particle had been marked in the collided particles array. If it has, then a search is performed against all other particles as well as those in the eight adjacent cells.

The new potential collisions that are found are added to an array in local memory. Once the search is complete, the elements that were used in that local array are moved into the list of potential collisions. This is done using the same type of procedure as in the initial finding routine. This involves a single atomic increment by to appropriate amount, followed by a loop copying up elements into the array of potential collisions.

It is worth nothing that we are not keeping a list of free nodes in the potential collision pool. This will lead to some waste, but at the current time it appears to be required to maintain performance and thread safety. We considered keeping a free list and had included it into a fair bit of the code, but then we ran into a thread safety problem. When we want to grab a new node we have to advance the first free reference to the next element. Because the reference to the first free element is shared across threads, this must be done using an atomic operation. That would lead to a line of code like the following.

```
int oldFF = atomic_xchg(ff,
  potentialCollisionPool[*ff].next);
```

Unfortunately, this isn't really thread safe. The reference to the first free value, `ff`, in the second argument to `atomic_xchg` is a read that isn't protected by the atomic call. That means that two threads could get the same value for `ff` before either one of them does the atomic exchange to update to the next one. The result would be two threads each holding the same node to store a potential collision in which would clearly lead to errors.

It is unclear at this time how critical a problem this is in general. For the tests presented here it was not a problem. Limiting the length of a time step can reduce the number of total collisions so that the potential collision pool doesn't overflow. However, it is likely that we need to find better ways to address this challenge. That will be the subject of further research.

Grid	N	Cells/N	OCL CPU	GPU1	GPU2	OMP CPU
$2k^2$	4m	1:1	630ms	1034ms	1117ms	727ms
$2k^2$	2m	2:1	235ms	530ms	842ms	315ms
$1k^2$	1m	1:1	81ms	223ms	137ms	189ms
$2k^2$	1m	4:1	120ms	375ms	737ms	163ms
$1k^2$	512k	2:1	43ms	126ms	79ms	85ms
$2k^2$	512k	8:1	95ms	318ms	721ms	99ms
$1k^2$	256k	4:1	27ms	80ms	62ms	42ms
$2k^2$	256k	16:1	49ms	267ms	692ms	84ms
$1k^2$	128k	8:1	15ms	64ms	57ms	27ms
$1k^2$	64k	16:1	10ms	64ms	48ms	19ms

Table 1

THIS TABLE SHOWS THE TIMING RESULTS FOR THE OPENCL CODE RUNNING ON BOTH A CPU AND TWO DIFFERENT GPUS COMPARED TO THE EXISTING, MULTITHREADED CODE ON A CPU FOR A VARIETY OF PARTICLE COUNTS AND GRID SIZES. FOR EACH CONFIGURATION, A WARM-UP SIMULATION WAS DONE FIRST, FOLLOWED BY FIVE RUNS THAT WERE AVERAGED TO GET THESE RESULTS.

4. Results

4.1 CPU/GPU Comparisons

The above algorithm was implemented in OpenCL for a 2-D system in which particles move in a straight line between collisions. We first tested the performance of this implementation on a PC with an NVIDIA GeForce GTX 670 graphics card (hereafter GPU1) and an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz CPU, and separately with a AMD Radeon HD 7970 (hereafter GPU2). The timing results are shown in table 1 and figure 1.

The simulations were set up with a square grid with grid cells that were one unit of length on each side. Particles 0.2 units in radius were placed in the simulation area uniformly with random velocities on the order of one unit. The simulation was run for one time step of 0.1 time units. For the simulations involving 1 million particles this set-up requires that nearly 1 million particle pairs be checked for collisions and over 20,000 potential collisions be added to the queues. Most of those potential collisions turn out to be real collisions that are actually processed.

When looking at the results, one should keep in mind that the OpenCL implementation has not yet been optimized,

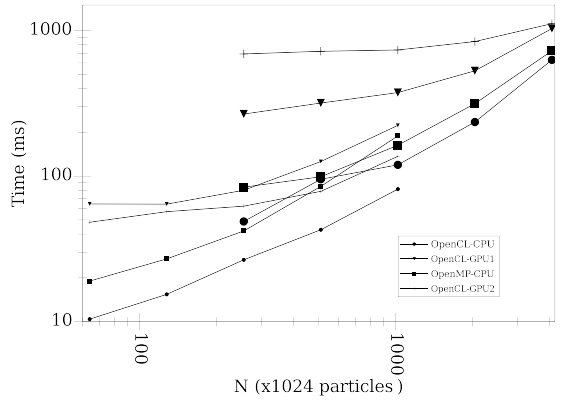

Fig. 1

THIS FIGURE SHOWS THE TIMING RESULTS GIVEN IN TABLE 1. SMALLER SYMBOLS ARE USED FOR THE RESULTS WITH THE GRID THAT HAS 1024 CELLS ON THE EDGE AND LARGER SYMBOLS ARE USED FOR THE GRID WITH 2048 CELLS ON THE EDGE. THE PLOT MAKES IT CLEAR THAT THE OPENCL IMPLEMENTATION OF THIS ALGORITHM IS FASTER ON THE CPU THAN OUR PREVIOUS OPENMP CODE. HOWEVER, THE GPU RESULTS GENERALLY LAG BEHIND BOTH CODES ON THE CPU. THE ONE EXCEPTION IS THE TWO LARGEST SIMULATIONS WITH THE SMALL GRID ON GPU2, WHICH OUTPERFORM THE OPENMP CODE.

especially for running on a GPU. On the other hand, the OpenMP implementation is one that we have had for a number of years and have been able to tweak for performance. Despite this, the OpenCL performs better on the CPU and is within a factor of several of the OpenMP code on the GPUs using a commodity gaming graphics card. The gap between OpenCL on the GPUs and the OpenMP code also closes as the particle count increases, with GPU2 beating OpenMP for the two largest particle counts using the smaller grid. For the largest simulations at each grid size, the difference is well below a factor of two on GPU1.

The table lists the ratio of number of grid cells to number of particles because empirical work on the original version of this code [9] found that for a fixed number of particles, the code gave optimal performance with a maximum ratio of 7:1. We have no yet performed a full set of tests to determine the optimal ratio for the OpenCL code, but these tests do show that this algorithm appears to have a greater cost for a larger number of grid cells, particularly on on the AMD GPU, GPU2. The OpenCL code is faster for using a smaller number of grid cells than a larger one for all particle counts. This isn't too surprising given how many of the passes launch threads based on the number of grid cells. As a result, it appears that the OpenCL code likely has an optimal ratio with fewer grid cells per particle. Resolving the ideal ratio is an area for future work.

The performance of the GPUs in these tests is consistent with the fact that the code has not yet been optimized for the GPU. Each thread group on a GPU is basically a SIMD unit that is most efficient when all the threads are doing the same thing, running a single instruction on multiple pieces of data. We have not yet optimized the code to try to keep threads doing the same thing. The kernels were written to contain as much work as was possible for a particular decomposition of the work without running into race conditions. The code likely needs to be refactored in a number of ways, including breaking up the kernels, to keep the work more consistent across the threads so that the GPUs can run more efficiently.

4.2 CPU Scaling

A second set of tests was run using only the CPU to look at how well this new algorithm, using OpenCL, scales relative to the old one using OpenMP. The results of these tests are shown in figure 2. These simulations were run on a Dell server with four 16-core Opteron 6272 processors, each with 16 GB of local RAM. So the machine has 64-cores and 64 GB or RAM total, allowing us to scale the simulations be to significantly larger than the earlier tests. The initial conditions were the same as before. For these simulations, the ratio of grid cells to particles was kept fixed at 1:1 for the OpenCL simulations and 4:1 for the OpenMP code. The number of cores used in the simulations was set at 16, 32, and 64 for each of the codes for a variety of particle counts. The timing results are shown in figure 2.

Both code scale remarkably linearly with particle count up to more than 67 million particles. Consistent with earlier results, the OpenCL code holds a reasonable speed advantage up to 4 million particles. Most of the advantage disappears at 16 million particles and there is remarkably little spread in runtime between the methods or the core counts in the largest simulations. More work needs to be done to determine why the OpenCL implementation loses ground for the largest simulations and what can be done, either in the code or the configuration, to prevent this.

4.3 Double Precision

These timing results came from code using single precision floating point values and arithmetic. Current GPUs tend to be much faster with single precision than double precision in most benchmarks. Unfortunately, there are some simulations that explore problems of scientific interest that require the additional precision. Hard sphere collisions for large N happen to be one of those problems. For that reason, we wanted to explore the impact of using double precision numbers here because this happens to be just such a situation.

The importance of double precision for these simulations was discovered when we put in a consistency check on the

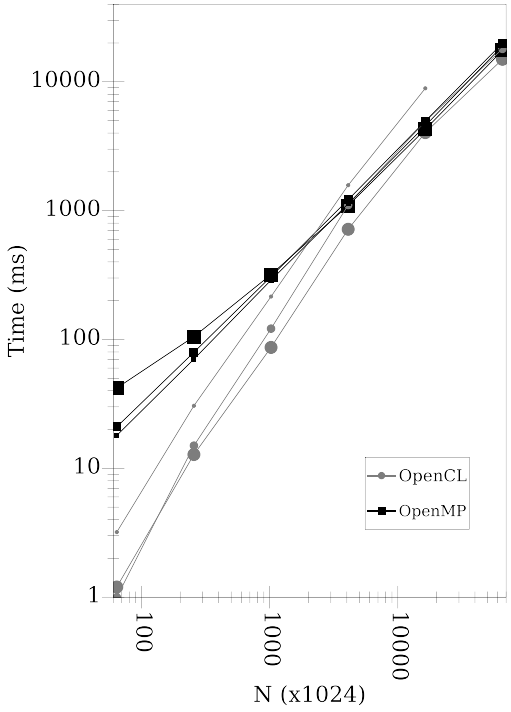

Fig. 2
THIS FIGURE SHOWS THE TIMING RESULTS ON A 64-CORE MACHINE COMPARING THE OPENCL IMPLEMENTATION OF THIS ALGORITHM WITH OUR EARLIER OPENMP CODE. FOR THESE SIMULATIONS, THE RATIO OF GRID CELLS TO PARTICLES WAS FIXED AT 1:1 FOR THE OPENCL CODE AND 4:1 FOR THE OPENMP CODE. THE SYMBOL SIZE INDICATES THE NUMBER OF CORES INVOLVED. SMALL SYMBOLS ARE 16 CORES, MEDIUM ARE 32, AND LARGE SYMBOLS ARE 64.

hard-sphere collisions. This check tested to see that when particles were advanced to the time of a potential collision that they were actually touching. This check failed when we required the separation between particles be within 1% of the sum of the particle radii. After double checking all the math and manually testing some of the values for the root finding, we realized that the problem was actually the accuracy of single precision floating point values. The simulation region was 1000 units across and the particles were 0.2 units in radius. That means that we were testing for an accuracy on the order of 10^{-5}. This is below the expected accuracy of single precision arithmetic, but would be met easily by double precision values.

Running a few tests showed us the very surprising result that changing the code to use the `double` type instead of `float` didn't have a significant impact on speed for our hardware. Our first assumption was that the cards, being commodity graphics cards and not cards specifically designed for HPC and GPGPU, were defaulting to single precision despite being told to use double precision. Attempts to measure the value of ϵ on the graphics cards showed that they were doing something different with `double` than with `float`, but the results those tests produced were not consistent with IEEE double precision numbers so more work is needed to determine what is happening in this area.

5. Conclusions

We found that a hard sphere collision algorithm using separate queues for each of many small spatial regions was implementable in OpenCL and that even without significant tuning, this code could outperform an earlier OpenMP implementation using a single queue when running on the CPU. Unfortunately, the GPU performance of the implementation currently lags behind that on the CPU.

There are many open questions and areas left for future work. In addition to those mentioned earlier in the paper, there are a few other areas that remain to be explored. First, does the difference in the ideal grid cell size alter how the performance varies with length of time step. The previous methods tend to scale the grid with the length of the time step. Because a simulation needs to go for a certain period of time, and the number of particles searched for collisions scales as the area of a grid cell, the total run time of a simulation tended to scale as $1/dt + dt^2$. This leads to an optimal time step that is fairly short to keep the cell sizes small. This work has already shown that this new algorithm performs better with larger grid cells. This could allow longer time steps to be taken, providing shorter total run times, even if the performance for a single time step is inferior.

This work looked only at the most basic of particle configurations where only collisions are considered. In realistic systems, the particle distribution is typically not so uniform and other forces, like gravity, can cause particles to clump. This alters the geometric distribution of the collisions. It is unclear at this time what impact that would have on this method.

Lastly, when the geometric distribution becomes significantly non-uniform, the spatial grid that was used here becomes ineffective. We have extended the older approach to use spatial trees as a more dynamic searching and locking data structure [8]. This same thing could be done on the GPU, though it is likely to be significantly more challenging to do so.

References

[1] Erich Elsen, Vaidyanathan Vishal, Mike Houston, Vijay S. Pande, Pat Hanrahan, and Eric Darve. N-body simulations on gpus. *CoRR*, abs/0706.3060, 2007.

[2] Simon Green. Particle simulation using cuda. *NVIDIA Whitepaper, December 2010*, 2010.

[3] K. A. Hawick and D. P. Playne. Hard-sphere collision simulations with multiple gpus, pcie extension buses and gpu-gpu communications. In *Proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing - Volume 127*, AusPDC '12, pages 13–22, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[4] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 123–131, New York, NY, USA, 2004. ACM.

[5] C. Lauterbach, Q. Mo, and D. Manocha. gproximity: Hierarchical gpu-based operations for collision and distance queries. *Computer Graphics Forum*, 29(2):419–428, 2010.

[6] Mark Lewis, Matthew Maly, and Berna L Massingill. Hybrid parallelization of n-body simulations involving collisions and self-gravity. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTAâĂŹ09)*, pages 324–330, Las Vegas, USA, July 2009. CSREA.

[7] Mark Lewis and Berna L Massingill. Multithreaded collision detection in java. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTAâĂŹ12)*, pages 583–592, Las Vegas, USA, July 2006. CSREA.

[8] Mark Lewis and Cameron Swords. Lock-graph: A tree-based locking method for parallel collision handling with diverse particle populations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'11, Volume I*, pages 157–161. CSREA Press, 2011.

[9] Mark C Lewis and Glen R Stewart. A new methodology for granular flow simulations of planetary rings-collision handling. In *Modelling and Simulation*, pages 292–297, 2003.

[10] Mark C Lewis and Nick Wing. A distributed methodology for hard sphere collisional simulations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications-Volume 1*, pages 404–409. CSREA Press, 2002.

[11] Berna L Massingill and Mark Lewis. Parallelizing a collisional simulation framework with plpp (pattern language for parallel programming). In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTAâĂŹ06)*, pages 608–614, Las Vegas, USA, July 2006. CSREA.

[12] Hammad Mazhar, Toby Heyn, and Dan Negrut. A scalable parallel method for large collision detection problems. *Multibody System Dynamics*, 26(1):37–55, 2011.

[13] D. P. Playne and K. A. Hawick. Classical mechanical hard-core particles simulated in a rigid enclosure using multi-gpu systems. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTAâĂŹ12)*, pages 76–82, Las Vegas, USA, 16-19 July 2012. CSREA.

[14] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.