

Tristan Shabbick

Professor Mordohai

CS 677 - Parallel Programming for Many Core Processors

13 May 2026

## Simple Projectile Simulator

### **Abstract**

Simulating large quantities of pairwise-interactive projectiles in real time is a computationally demanding challenge with direct relevance and impact in the game development sphere, of particular pertinence to the bullet-hell genre. This report compares four approaches to pairwise gravity projectile simulation, including a naive implementation on the CPU (Central Processing Unit), a naive GPU (Graphics Processing Unit) implementation, a GPU implementation with spatial binning, and finally an “optimized” GPU implementation that combines spatial binning with taking advantage of GPU’s memory hierarchy. Results demonstrate that GPU acceleration, even in its most naive form, offers dramatic performance advantages that are otherwise untouchable for CPU-based simulations, and that specialized techniques and targeted memory optimizations can yield further, meaningful gains. All programming was done in NVIDIA CUDA and mentions of terms such as shared memory, blocks, threads, etc., are explicitly in reference to CUDA and/or NVIDIA’s GPU architectures.

### **Introduction**

Real-time physics simulations involving large numbers of entities are a long-standing challenge in interactive software. In particular, bullet-hell style video games (a subgenre characterized by dense fields of simultaneously active projectiles) place extreme demands on simulation throughput in order to maintain playability from the point of view of users. Even modest projectile counts in the low thousands can easily overwhelm a single-threaded CPU approach when each projectile must account for local gravitational attraction from every other projectile or detect collisions (this report will largely focus on the effects of pairwise gravity between projectiles, with significantly less emphasis on collision detection).

The core bottleneck stems from the nature of pairwise gravitational interaction. For  $n$  active projectiles, a straightforward (“naive”) gravity simulation requires each projectile to consider the influence of every other, resulting in an algorithmic complexity of  $O(n^2)$ . At  $n = 4096$ , this is already roughly 16 million pairwise computations per frame. Reaching a target of  $n = 100,000$  projectiles while maintaining an enjoyable framerate is effectively impossible on the CPU.

As a result, compute devices designed for massive parallelism, such as GPUs, are highly effective for workloads in which individual tasks can be executed independently. Updating the state of each projectile satisfies this criterion well: each projectile’s new position and velocity (as affected by gravity of other projectiles) can be computed without waiting on any other projectiles’ result; in the context of CUDA, this essentially means there is a 1:1 mapping between threads on the device and projectiles, where each thread is directly responsible for updating exactly one projectile.

Additional notes: OpenGL (glew, glfw), OpenCV, etc., were used to visualize the simulation on Windows 11. Calls to OpenGL’s render pipeline to put projectiles on screen, the spawning of new projectiles, keeping track of how many projectiles are active, etc., take place exclusively on the CPU in all versions of the application, and may contribute to encroaching bottlenecks with sufficiently many projectiles to render and/or keep track of.

## **The Naive CPU Implementation**

The CPU implementation follows the most straightforward approach: at each simulation step, a snapshot of all active projectiles is copied into a read-only buffer (to prevent race conditions when updating projectiles’ velocity). An outer loop iterates over every active projectile  $i$ . For each  $i$ , an inner loop iterates over every other active projectile  $j$  to accumulate the gravitational force exerted on  $i$  (per the inverse square law). After the inner loop completes,  $i$ ’s velocity and position are updated using the accumulated gravity vector as well as its standard motion parameters (which for the purposes of this project, include but are not limited to velocity, amplitude, frequency, phase, etc.). Then, the projectile’s updated state is written back to the main buffer. As such, this algorithm runs in  $O(n^2)$  time. Its simplicity and nature of being on the CPU

means that there is essentially no overhead outside the core computation, making it a useful baseline.

Performance wise, at  $n = 1024$  projectiles, the CPU implementation achieved approximately 240 frames per second (FPS) average, which is beyond service for real-time use; however, at  $n = 4096$ , a four fold increase, average FPS plummets sixteen times downwards to an unplayable  $\sim 15.6$  FPS, whose scaling is exactly what is expected given the  $O(n^2)$  time complexity. Additional CPU performance data:

| # Projectiles | CPU Implementation      |
|---------------|-------------------------|
|               | Avg FPS-CPU (i9 13900k) |
| 2048          | 62.73                   |
| 4096          | 15.94                   |
| 8192          | 5.48                    |
| 10240         | 4.21                    |
| 65536         | N/A                     |

(Note: Program would not start at  $n = 65536$ )

Overall, the naive CPU implementation performs admirably so long as the number of projectiles is relatively small (less than or equal to 2048, using an average of 60 FPS as the near-universal standard for a “good experience”). Though, reaching upwards into the scale of tens of thousands of projectiles and beyond, requires a fundamentally different approach.

### Naive GPU Implementation

The naive GPU implementation ports the same approach as the naive CPU implementation above to NVIDIA’s proprietary parallel compute language, CUDA. A single kernel is launched with one thread per projectile (with as many blocks as necessary to cover every projectile; thread count is 128 by default, but is configurable through the command line). Each thread reads the projectile snapshot from global memory, iterates over all other projectiles in global memory to compute the gravitational influence, updates the projectile’s velocity and position, and writes back the updated state to global memory.

Simply porting the naive algorithm to the GPU yielded the following performance results:

| # Projectiles | Naive GPU Implementation |                      |
|---------------|--------------------------|----------------------|
|               | Avg FPS                  | Avg Kernel Time (ms) |
| 2048          | 1000.52                  | 0.0016               |
| 4096          | 479.02                   | 0.0079               |
| 8192          | 236.59                   | 0.0172               |
| 10240         | 200.8                    | 0.0199               |
| 65536         | 31.95                    | 0.1945               |

The performance gains over the CPU are substantial. At  $n = 2,048$ , average FPS rises from 62.73 on the CPU to 1,000.52 on the GPU, nearly a 16-times improvement. At  $n = 4096$ , the GPU achieves 479.02 FPS against the CPU's 15.94. The GPU continues to sustain usable frame rates and predictably scales far better than the CPU version, as the GPU only requires  $n$  threads, and each thread only does  $O(n)$  work.

It is worth noting and potentially exploring further that at default NVIDIA CUDA compiler (nvcc) settings, the naive GPU kernel achieves 83.3% occupancy as measured by a CUDA's `cudaOccupancyMaxActiveBlocksPerMultiprocessor()` function. This can be improved by *decreasing* the amount of registers each thread has access to with the `-maxrregcount` compiler flag, from 255 (default) to 40 or 32 (other valued untested or yielded unchanged / worse occupancy), which raises the kernel's occupancy on the GPU to 100% and yields approximately a 5.5% performance improvement. This is likely due to the fact that the kernel is memory-latency bound — threads spend most of their time waiting on global memory reads rather than executing arithmetic. Reducing the register count per thread allows for more streaming multiprocessors (SMs) to be present and executing instructions on the GPU, which naturally hides latency better. In other words, more threads spilling to their local memory has better throughput than fewer threads, not spilling, but instead waiting on global memory.

Register testing data:

| <b>Naive GPU Implementation @ 8192 Projectiles</b> |                |                    |                  |
|----------------------------------------------------|----------------|--------------------|------------------|
| <b># Registers</b>                                 | <b>Avg FPS</b> | <b>Kernel Time</b> | <b>Occupancy</b> |
| 32                                                 | 247.42         | 0.0085             | 100%             |
| 40                                                 | 251.1          | 0.0083             | 100%             |
| (default) 255                                      | 236.59         | 0.0172             | 83%              |

### Naive Binning GPU Implementation

Despite the impressive performance gains obtained by switching to a GPU-based solution, the ported naive implementation still requires each thread to iterate over all  $n$  projectiles in global memory. This pattern is particularly well-suited for optimization through spatial binning, which aims to reduce a thread's workload by limiting each projectile's gravity search to a local neighborhood.

This is achieved by first dividing the simulation space (screen space) into a grid of bins, which in all tested configurations, is a 32 by 18 arrangement (to align well with a traditional 16:9 aspect ratio). Then, a pre-processing kernel assigns each projectile to its corresponding bin based on its position on screen. A second kernel then updates each projectile: blocks are mapped one-to-one with bins with as many threads per block as the max bin size (not overridden by command line options to specify thread count); each thread within a block searches only its own bin and its adjacent bins, limiting the search to at most  $9 * \text{max bin size}$ , as opposed to all  $n$  projectiles.

Important note: binning introduces an important simulation accuracy tradeoff in the case of pairwise gravity: Newton's law of universal gravitation states that each object has an attractive force to every other object. By artificially limiting the range of this effect, the influence of gravity, no matter how strong, can only go so far, this is a limitation of binning itself (it is worth noting that there is no such accuracy tradeoff when performing collision detection, as only adjacent bins have a chance to be relevant). Additionally, in this particular binning implementation, in scenarios where many projectiles attract each other and cluster tightly, bins fill quickly. If a projectile arrives in a full bin, it is excluded from the gravity computation and

updates using its last computed velocity. Attempted solutions to this problem will be described later in this report.

As for performance, binning yields consistent improvements across all tested projectile counts; ranging from a 37% uplift in the worst case to a 79% increase in the best case, at 2,048 and 8,192 projectiles respectively. Additional performance data can be viewed below:

| # Projectiles | Naive Binning GPU Impl. |                      |
|---------------|-------------------------|----------------------|
|               | Avg FPS                 | Avg Kernel Time (ms) |
| 2048          | 1372.82                 | 0.0029               |
| 4096          | 746.14                  | 0.0033               |
| 8192          | 423.57                  | 0.0075               |
| 10240         | 339.51                  | 0.0098               |
| 65536         | 52.27                   | 0.0186               |

Note: adjustments using the `-maxrregcount` compiler flag did not yield any noticeable performance gain here or in future iterations, as such, there is no worthwhile data to display. Occupancy did increase with fewer registers as it did before, but no measurable performance increase or decrease was present.

### Optimized Binning GPU Implementation

This final implementation builds on the naive binning approach, though now with the addition of shared memory utilization, float4 data packing (as the standard data structure to represent a projectile was far too large to use with limited shared memory), and constant memory for the binning parameters. The simulation's update pipeline now expands to three kernels, the first, just as before, bins the projectiles, the second computes the gravitational influence using shared memory, and the third performs standard velocity and position updates. The former and latter kernels are rather straightforward and as such they will not be expanded upon further in this report.

The kernel responsible for the gravity computation operates by having thread 0 first configuring and setting up the neighborhood of bins, followed by threads 0 to 8 copying a bin from global memory into shared memory, and the remaining threads each attempt to take a

projectile from its bin and search its neighboring bins. This approach has a global / shared memory read ratio of the configured bin size, which by default and what was used in testing is a ratio of 128. The occupancy of this kernel is not ideal, hovering around 40% (as a result of the strict shared memory limits due to the size of the bins), but despite this lower occupancy, there are still tangible performance benefits:

| # Projectiles | Shared + Constant Binning GPU Impl. |                      |
|---------------|-------------------------------------|----------------------|
|               | Avg FPS                             | Avg Kernel Time (ms) |
| 2048          | 1450.71                             | 0.0014               |
| 4096          | 784.4                               | 0.0021               |
| 8192          | 441.45                              | 0.0035               |
| 10240         | 362.54                              | 0.0042               |
| 65536         | 56.02                               | 0.0062               |

These results showcase an approximately a 6.5% performance uplift compared to the previous kernel, whose occupancy was significantly higher, ranging from 75% to 100%. Compared to the CPU baseline, the optimized binning GPU implementation achieves an average improvement of approximately 59 times, with best showings boasting an ~86 times performance improvement with far superior asymptotic scaling as  $n$  increases.

### Cross-Device Observations

All three GPU kernel implementations were benchmarked on two different systems, one with an NVIDIA RTX 4090, the other with an NVIDIA RTX 3070 Ti (Mobile). The benchmark data is as follows:

| GPU Used -><br>Kernel Tested @ 8192 Projectiles | RTX 3070 Ti Mobile |                      | Performance Improvement Over Previous Kernel<br>(Outer = Avg Fps, Inner = Kernel Time) |      |      |      | RTX 4090 |                      |
|-------------------------------------------------|--------------------|----------------------|----------------------------------------------------------------------------------------|------|------|------|----------|----------------------|
|                                                 | Avg FPS            | Avg Kernel Time (ms) |                                                                                        |      |      |      | Avg FPS  | Avg Kernel Time (ms) |
| Naive GPU                                       | 103.23             | 0.1021               | N/A                                                                                    | N/A  | N/A  | N/A  | 253.68   | 0.0274               |
| Naive Bin.                                      | 284.55             | 0.0369               | 1.76                                                                                   | 1.77 | 2.65 | 0.67 | 423.57   | 0.0075               |
| Shared Bin.                                     | 325.54             | 0.0062               | 0.14                                                                                   | 4.95 | 1.1  | 0.04 | 441.45   | 0.0035               |

This testing revealed that the binning kernel that took advantage of shared memory (plus other optimizations) seems to provide a proportionally greater benefit on less capable hardware compared to the naive binning kernel. The working hypothesis is that the RTX 4090's exceptionally large L2 cache (72 MB, versus 4 MB on the 3070 Ti (Mobile)) and much wider memory bus (384-bit at 1.01 TB/s versus 256-bit at 448 GB/s) allow it to effectively cache the

frequently accessed projectile data in global memory, partially negating the advantage of explicit shared memory management.

### **Addressing Full Bins**

In both binning implementations, before anything else, bins are created and allocated with a capacity for as many projectiles as was specified in the command line, or by the defaults. Projectiles are then sorted into these bins based on their position. As mentioned above, this can lead to a situation where bins are full where projectiles are not associated with any bins (overflow bins were not implemented here), and thus, are not affected by gravity whatsoever. An attempt was made to rectify this by modifying the update pipeline with a sorting-based approach. Each projectile gets mapped to a bin as before, but now with an auxiliary array of length  $n$  (to house every possible projectile) which contains two fields: the projectile index and what bin it belongs to. This array is then sorted on the GPU using parallel Radix Sort (from the CUB library) by bin, and then scanned once more to record the index at which each bin starts and stops. On the GPU, each thread is assigned one projectile and uses that projectile's position to compute which bin it belongs to. It then looks up the 9 surrounding bins in the range map and iterates over only the projectiles those bins contain, rather than the entire array. In practice this resolves the bin overflowing problem; however, it comes with a steep performance penalty (likely as a result of the sorting every frame before updating), often performing worse than the naive, non-binning GPU implementation.

### **Miscellaneous Optimization Attempts**

Keeping track of how many projectiles are active is necessary for knowing when old projectiles in the main array can be cleared and fitted with a newly spawned projectile's data. All implementations keep projectile state entirely on the GPU, with this being the exception, active projectiles are computed with a simple for loop on the CPU side after the update on the GPU, as using `cudaMemcpy` takes far too slow to keep track of a single integer. That said, an OpenMP parallel reduction was used in an attempt to possibly speed up keeping track of active projectiles

on the CPU, but unfortunately led to staggering performance losses, even with  $n$  as high as 65,536, likely due to overhead.

Approximate math functions were used throughout, as the additional accuracy has no perceivable effect on the simulation, thus providing an essentially free speedup.