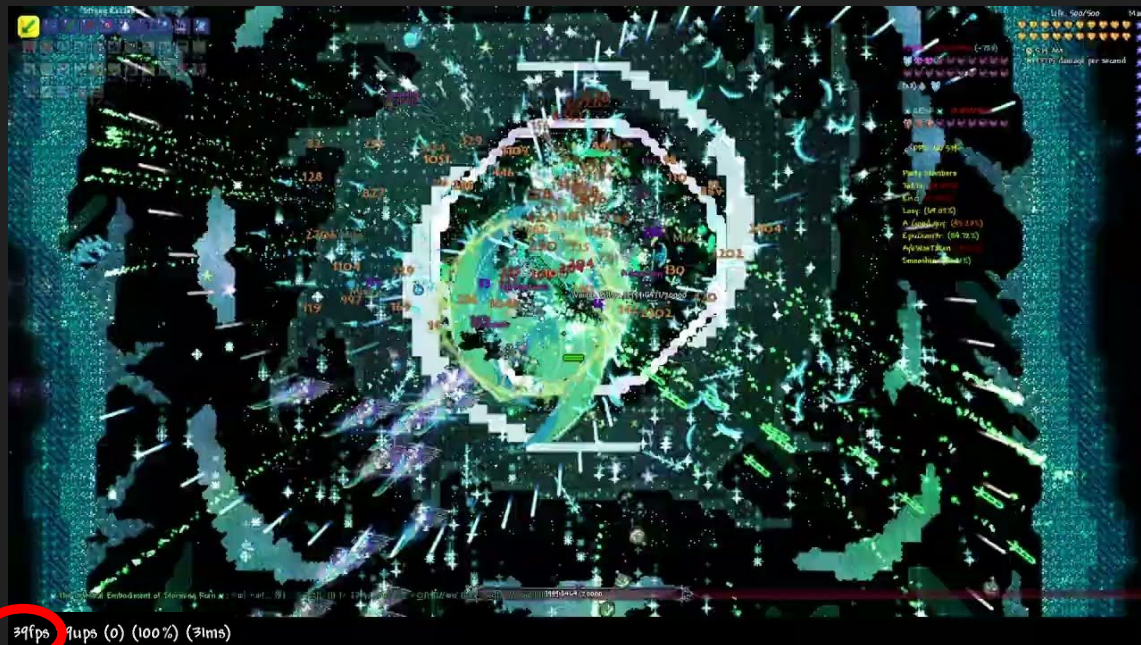


Simple Projectile Simulator

Tristan Shabbick

The Problem & Its Importance

- Handling projectile updates by the thousands is expensive!
 - Bullet-hell games are in shambles!
- Even more so when they do more than just move in a straight line:
 - Local Gravity
 - Collision Detection



Naive CPU Implementation



- The process:
- For each projectile (n)
 - For each *other* projectile (n-1)
 - Calculate how gravity affects us
 - Update position
- Runs in $O(n^2)$ time
- Works okay for small n...

```
void
update_cpu(Projectile* h_projs, float delta_t)
{
    /* needs a read-only buffer to avoid race conditions when updating position,
    | etc. */
    Projectile snapshot[MAX_PROJS];
    memcpy(snapshot, h_projs, MAX_PROJS * sizeof(Projectile));

    for (int i = 0; i < MAX_PROJS; i++)
    {
        if (!snapshot[i].active)
            continue;

        Projectile p = snapshot[i];

        float normalized_dt = delta_t * 60.0f;

        float grav_x = 0.0f;
        float grav_y = 0.0f;

        /* gravity computation */
        for (int j = 0; j < MAX_PROJS; j++)
        {
            if (j == i || !snapshot[j].active || snapshot[j].gravity == 0)
                continue;

            float dx = snapshot[j].pos.x - p.pos.x;
            float dy = snapshot[j].pos.y - p.pos.y;
            float dist_sq = dx * dx + dy * dy;

            if (dist_sq < 1.0f)
                continue;

            float dist = sqrtf(dist_sq);
            float force = snapshot[j].gravity / dist_sq;

            grav_x += (dx / dist) * force;
            grav_y += (dy / dist) * force;
        }

        update_standard_projectile_velocity(&p, grav_x, grav_y, delta_t,
                                           normalized_dt);

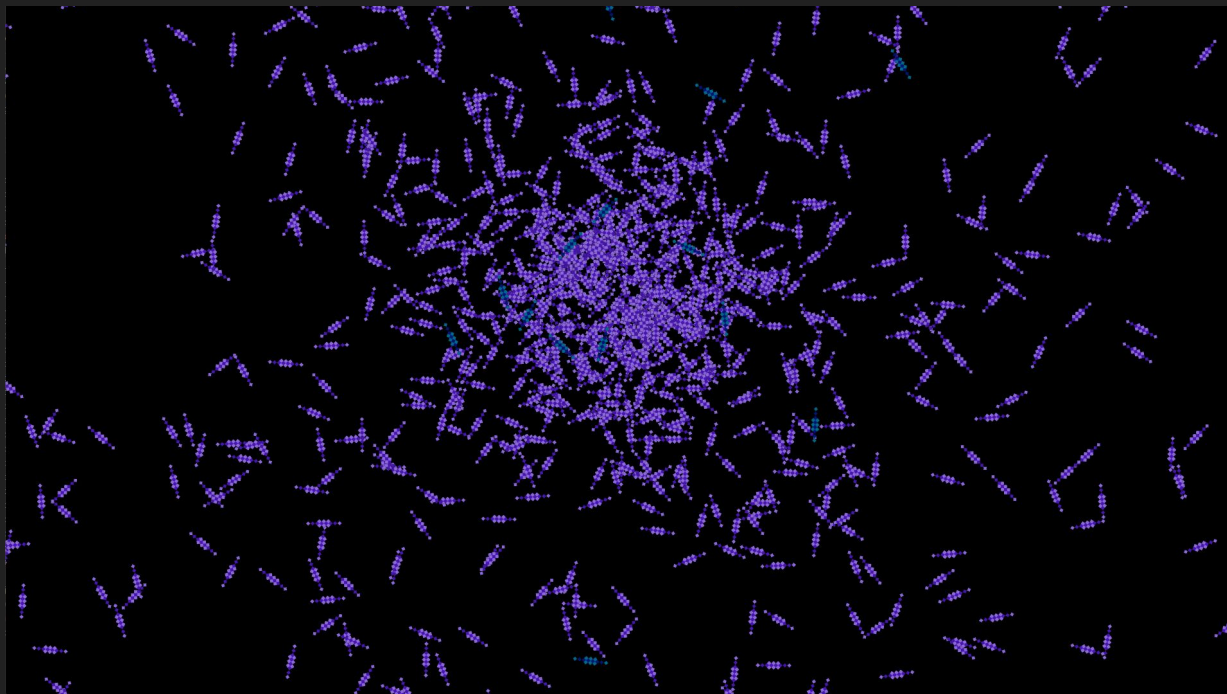
        h_projs[i] = p;
    }
}
```

Naive CPU Implementation



- Fast CPU and small n ? Completely usable, even at $O(n^2)$!

$n = 1024$



FPS: 233.22

Avg: 240.86

Active: 1024

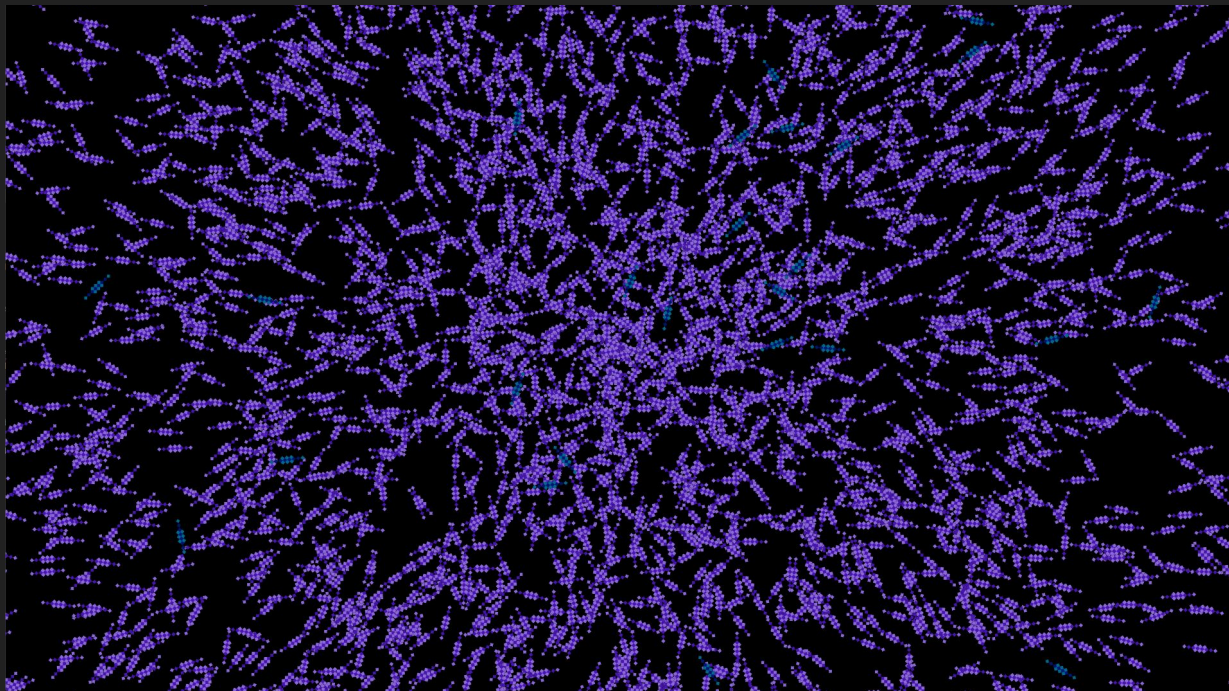
Max: 1024

Naive CPU Implementation



- Fast CPU and not as small n ? Completely *unusable*...

$n = 4096$



FPS: 15.62

Avg: 15.63

Active: 4096

Max: 4096

n increased by 4x

FPS decreased by
16x!

Naive CPU Implementation



- The process: --->
- For each projectile (n)
 - For each *other* projectile (n-1)
 - Calculate how gravity affects us
 - Update position
- $O(n^2)$
- Works okay for small n... (kind of)
- We want n to be as large as 100,000!
 - CPU is no longer an option :(
- We turn to the GPU...

```
void
update_cpu(Projectile* h_projs, float delta_t)
{
    /* needs a read-only buffer to avoid race conditions when updating position,
    | etc. */
    Projectile snapshot[MAX_PROJS];
    memcpy(snapshot, h_projs, MAX_PROJS * sizeof(Projectile));

    for (int i = 0; i < MAX_PROJS; i++)
    {
        if (!snapshot[i].active)
            continue;

        Projectile p = snapshot[i];

        float normalized_dt = delta_t * 60.0f;

        float grav_x = 0.0f;
        float grav_y = 0.0f;

        /* gravity computation */
        for (int j = 0; j < MAX_PROJS; j++)
        {
            if (j == i || !snapshot[j].active || snapshot[j].gravity == 0)
                continue;

            float dx = snapshot[j].pos.x - p.pos.x;
            float dy = snapshot[j].pos.y - p.pos.y;
            float dist_sq = dx * dx + dy * dy;

            if (dist_sq < 1.0f)
                continue;

            float dist = sqrtf(dist_sq);
            float force = snapshot[j].gravity / dist_sq;

            grav_x += (dx / dist) * force;
            grav_y += (dy / dist) * force;
        }

        update_standard_projectile_velocity(&p, grav_x, grav_y, delta_t,
                                           normalized_dt);

        h_projs[i] = p;
    }
}
```

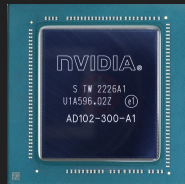
Suitability for GPU Acceleration?

- Fortunately, updating logic **nearly 100% parallelizable!**
 - Every projectile can be updated by a single thread on the GPU
- Logic that isn't suitable goes on the **CPU**:
 - Spawning new projectiles
 - Keeping track of how many are active
 - Faster than keeping track on the GPU
 - Calls to the OpenGL render pipeline to put the projectiles on the screen
 - At some point these will become a bottleneck for the GPU

A photograph of an NVIDIA GeForce 7300 GS graphics card. The card is blue and features the NVIDIA logo at the top. Below the logo, the text "S TW 2226A1", "U1A596.02Z", and "AD102-300-A1" are printed. A small circular logo with the letters "CI" is also visible. The card is shown from a top-down perspective, highlighting its compact design and the gold-plated pins on the PCI Express connector.

- Occupancy: 83.3% `cudaOccupancyMaxActiveBlocksPerMultiprocessor()`
- Ideal performance: # Threads $64 \leq n \leq 192$
 - Something else we can tweak too... registers!

Naive GPU Implementation



- Goal: “make it work on the GPU”
 - Global memory only, no fancy optimization strategies e.g., binning, spatial sorting, etc.
 -
- Each thread runs the gravity computation
 - n threads each doing $O(n)$ work
 - Scales far better than the CPU version:

# Projectiles	CPU Implementation	Naive GPU Implementation	
	Avg FPS-CPU (i9 13900k)	Avg FPS	Avg Kernel Time (ms)
2048	62.73	1000.52	0.0016
4096	15.94	479.02	0.0079
8192	5.48	236.59	0.0172
10240	4.21	200.8	0.0199
65536	N/A	31.95	0.1945

- Occupancy: 83.3% `cudaOccupancyMaxActiveBlocksPerMultiprocessor()`
- Ideal performance: # Threads $64 \geq n \leq 192$
 - Something else we can tweak too... registers!
- **nvcc compiler flag: `-maxrregcount=...`**
 - Some performance to be squeezed out --->

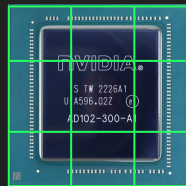
● But Why/How?

- The kernel is memory latency bound
 - (Would probably be catastrophic if it was compute bound)
- Waiting on global < Spilling to local
- Allows for more SMs at once = better latency hiding

~5.5% performance gain

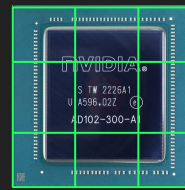
Naive GPU Implementation @ 8192 Projectiles			
# Registers	Avg FPS	Kernel Time	Occupancy
32	247.42	0.0085	100%
40	251.1	0.0083	100%
(default) 255	236.59	0.0172	83%

Naive Binning GPU Implementation

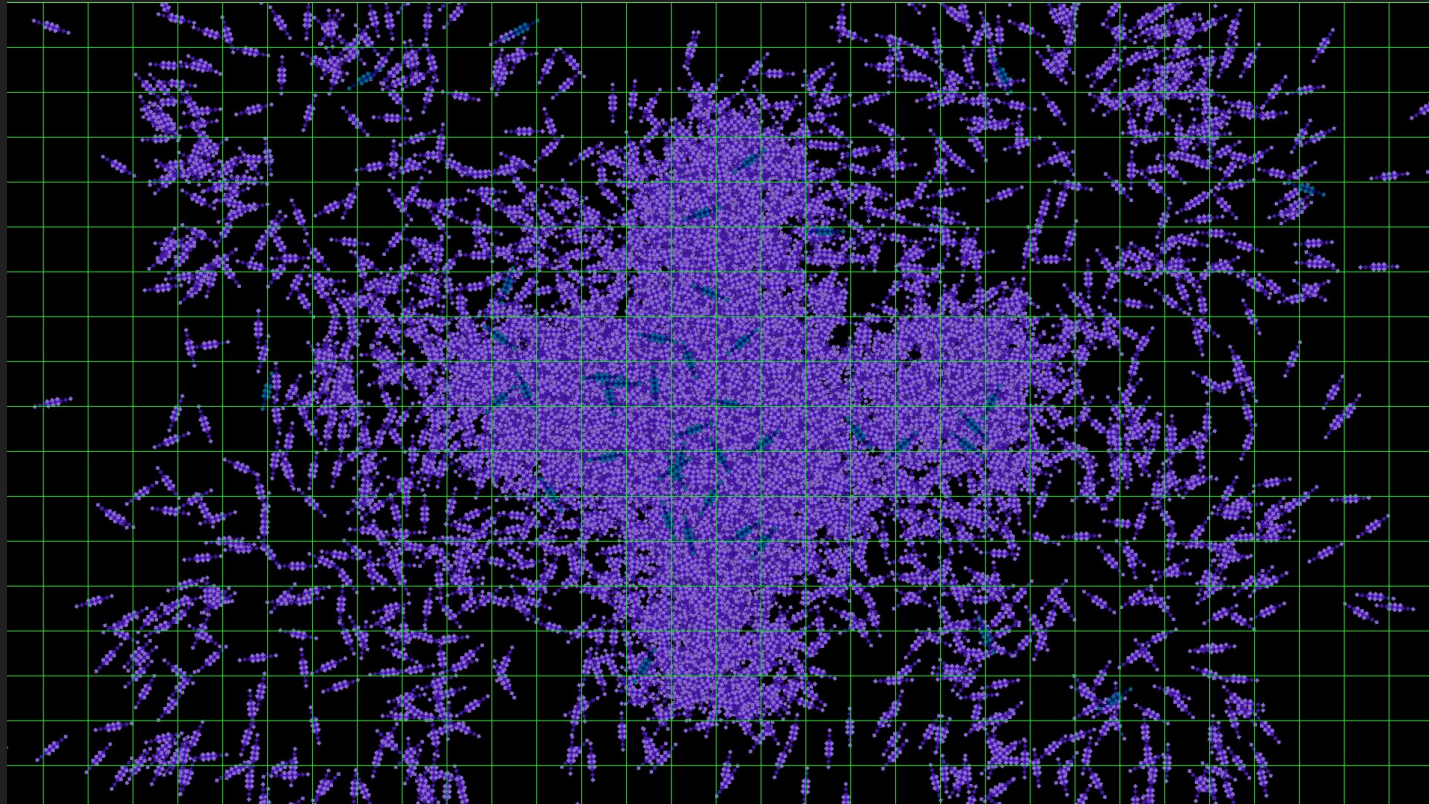


- Goal: “place projectiles into bins so that each projectile only needs to search it’s neighboring bins”
 - Makes much more sense for collision detection than gravity
 - Still global memory only at this point
- First kernel places each projectile into a bin
- Second kernel updates each projectile
 - 1:1 mapping blocks to bins
 - All threads in a block only search a 3x3 neighborhood (self + 8 adjacent bins)
 - $9 * \text{BIN_SIZE}$ searches instead of MAX_PROJS per thread at most!
- Small problem
 - When you have projectiles that attract each other, bins tend to get full quickly!
 - If a projectile lands in a full bin, it isn’t affected by gravity; only updated by last known velocity

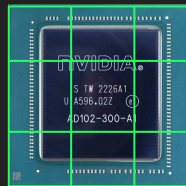
Naive Binning GPU Implementation



- Visualization: (32 x 18 bins)



Naive Binning GPU Implementation

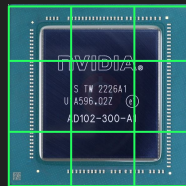


- Performance!
- Important Note: Binning necessarily changes program behavior
 - Less ideal for a realistic gravity simulation; generally not an issue for collision detection

# Projectiles	CPU Implementation	Naive GPU Implementation		Naive Binning GPU Impl.	
	Avg FPS-CPU (i9 13900k)	Avg FPS	Avg Kernel Time (ms)	Avg FPS	Avg Kernel Time (ms)
2048	62.73	1000.52	0.0016	1372.82	0.0029
4096	15.94	479.02	0.0079	746.14	0.0033
8192	5.48	236.59	0.0172	423.57	0.0075
10240	4.21	200.8	0.0199	339.51	0.0098
65536	N/A	31.95	0.1945	52.27	0.0186

- $1000.52 \rightarrow 1372.82 = \sim 37.2\%$ increase
- $479.02 \rightarrow 747.14 = \sim 55.7\%$ increase
- $236.59 \rightarrow 423.57 = \sim 79\%$ increase
- $200.8 \rightarrow 339.51 = \sim 69.1\%$ increase
- $31.95 \rightarrow 52.27 = \sim 63.6\%$ increase (likely starting to run into compute, memory latency, and/or CPU limitations)

Naive Binning GPU Implementation

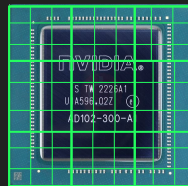


- Binning Kernel Occupancy: 100%
- Update Bins Kernel Occupancy: 75% (by default)
 - Adjusting max registers doesn't have much, if any, impact:

Naive Binning GPU Impl. @ 8192 Projectiles			
# Registers	Avg FPS	Kernel Time	Occupancy
32	420.13	0.0039	100%
40	421.26	0.0103	100%
48	407.33	0.0096	83%
(default) 255	423.57	0.0075	75%

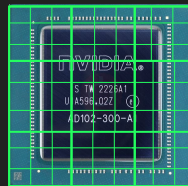
- Could be a result of compute time drastically decreasing over previous kernel (as the register delta likely wouldn't change much)
- cudaMemcpy / cudaMemcpySet's and 2x kernel launches are likely more of a bind

Optimized Binning GPU Implementation



- Goal: Simple, “make the previous kernel / Naive Binning on the GPU run faster!”
- Three Kernels:
 - First places each projectile into a bin
 - Second performs the gravity influence
 - Third performs the standard velocity updating
- Shared Memory!
 - Every block loads its corresponding bin and its 8 adjacent bins into shared memory
 - Thread 0 sets up the neighborhood of bins
 - Threads 0 to 8 copy a bin from global into shared
 - Threads 0 to #-of -projectiles-in-the-bin-minus-one each take a projectile from its bin and apply gravity by searching its / neighboring bins

Optimized Binning GPU Implementation



- Shared Memory!

- Every block loads its corresponding bin and its 8 adjacent bins into shared memory
 - Thread 0 sets up the neighborhood of bins
 - Threads 0 to 8 copy a bin from global into shared
 - Threads 0 to #-of -projectiles-in-the-bin-minus-one each take a projectile from its bin and apply gravity by searching its / neighboring bins
- Float4 packing — the ‘projectile’ struct is too large for shared memory ->
 - Gravity computation only needs: pos.x, pos.y, gravity, & index
 - 72 bytes -> 16 (w/ alignment) (should also help with coalesced access times)

- Occupancy

- Binning Kernel: 100%
- Gravity Kernel: 41.7% (by default; BIN_SIZE = 128 -- shared memory bound!)
 - BIN_SIZE = 340 -> 44.3% occupancy; 166 -> 43.2% occupancy; 132 -> 43% occupancy
- Velocity Kernel: 100%

- Constant Memory

- Binning specifications copied over

```
typedef struct
{
    uint8_t type;
    uint8_t tracked;
    uint8_t active;

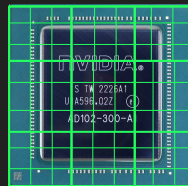
    float rotation;
    float gravity;

    float lifetime;
    float max_lifetime;

    float amplitude;
    float frequency;
    float phase;
    float wave_phase;
    float2 origin;

    float2 pos;
    float2 old_pos;
    float2 velocity;
} Projectile;
```

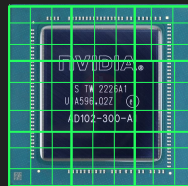
Optimized Binning GPU Implementation



- Shared Memory Cont.

- Assuming default `bin_size` of 128 (`bin_size` dictates `thread_count` for gravity update kernel, for the mapping)
 - And assuming each bin is 100% full
- Each block performs $9 * 128 * 4 = 4,608$ global reads to populate the shared memory
- Each block then accesses $9 * 128 * 128 * 4 = 589,824$ from shared memory
 - 128x ratio -> `BIN_SIZE` ratio
 - “Each of the 128 threads checks 9 bins with each 128 projectiles and loads from the packed float4”

Optimized Binning GPU Implementation



- Results!

# Projectiles	CPU Implementation	Naive GPU Implementation		Naive Binning GPU Impl.		Shared + Constant Binning GPU Impl.	
	Avg FPS-CPU (i9 13900k)	Avg FPS	Avg Kernel Time (ms)	Avg FPS	Avg Kernel Time (ms)	Avg FPS	Avg Kernel Time (ms)
2048	62.73	1000.52	0.0016	1372.82	0.0029	1450.71	0.0014
4096	15.94	479.02	0.0079	746.14	0.0033	784.4	0.0021
8192	5.48	236.59	0.0172	423.57	0.0075	441.45	0.0035
10240	4.21	200.8	0.0199	339.51	0.0098	362.54	0.0042
65536	N/A	31.95	0.1945	52.27	0.0186	56.02	0.0062

- ~6.5% improvement over previous kernel
- Different Bin Sizes tested:

Bin Size	Shared + Constant Binning GPU Impl. @ 8192 Projectiles		
	Avg FPS	Avg Kernel Time (ms)	Occupancy
64	406.31	0.0028	37.50%
128	420.78	0.0036	41.70%
256	399.94	0.0046	33.33%
340	428.61	0.0028	44.30%

- Overall Performance improvement over CPU: 5874.98% on average
 - 62.73 -> 1450.71 = ~23x; 4.21 -> 362.54 = ~86x (scales better with more projectiles)

Extraneous Experiments, Testing, & Reproduction

- All testing done on an Intel i9 13900k / RTX 4090 system with the following arguments:

- `./demo -c 72 -s 90 -a 1.5 -f 0.05 -g 5 -G 60 -l 960 -test [-cpu | -K 0 | -K 1 | -K 2]`
 - Add -r 500 for tests ≥ 65536 projectiles

- Attempted using OpenMP to speedup --->

- Led to a *significant* FPS penalty, >25%, due to

- Attempted Spatial Sort-ing on Bins

- Wanted to avoid situation where bins fill up and gravity
- Used Radix Sort from the CUB library, but its (the entire approach) implementation was foregone as it was convoluted and significantly slower
 - Not a bad tradeoff if simulation accuracy is a real concern

```
#pragma omp parallel for reduction(+ : active_count)
for (int i = 0; i < MAX_PROJS; i++)
{
    if (h_projs[i].active)
    {
        active_count++;
    }
}
```

- Approximate Math functions used throughout

- Free gains

Extra Extraneous Experiments

GPU Used ->	RTX 3070 Ti Mobile		Performance Improvement Over Previous Kernel				RTX 4090	
Kernel Tested @ 8192 Projectiles	Avg FPS	Avg Kernel Time (ms)	(Outer = Avg Fps, Inner = Kernel Time)				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

- The Kernel using shared memory scales significantly better on less capable devices, why?
- Working Theory: Hardware Difference?

- RTX 3070 Ti (Mobile)
 - 256-bit memory bus width
 - 448.0 GB/s bandwidth
 - 128 KB L1 cache (per SM)
 - 4 MB shared L2 cache

- RTX 4090
 - 384-bit memory bus width
 - 1.01 TB/s bandwidth
 - 128 KB L1 cache (per SM)
 - 72 MB shared L2 cache

- The 4090 may just be able to brute force with its ridiculously large L2 cache and much larger and faster memory bus

Simple Projectile Simulator

Demo

Simple Projectile Simulator

Tristan Shabbick

Questions & Comments