CUDA Point-Plane Collisions

The goal is to maximize particles possible in a physics simulation with 6 plane checks and basic gravity while minimizing kernel and thus convergence time.

Linux

Dependencies

For code that is run on the Linux machine for access to a 4090, this is what was necessary:

1. To get the VM CUDA to run, append

```
export PATH=/usr/local/cuda-11.7/bin$PATH
```

to `~/.profile` and `source ~/.bashrc`. To know it works, run `nvcc --version`.

2. Install GLM somewhere - I added an environment variable `export GLM_INCLUDE_DIR=~/packages/glm` to my `.profile` and got the code from here. Feel free to just modify `CMakeLists.txt` to point to your glm installation if its local to the project, or elsewhere.
3. `CMakeLists.txt` needed to be modified to find the header files in `/usr/` using this.
4. To run `ncu` meaningfully, you need root access. https://developer.nvidia.com/nvidia-development-tools-solutions-err_nvgpuctrperm-permission-issue-performance-counters

Helpful Commands

- `cat /etc/os-release` shows the distro and device architecture.
- `nvidia-smi` provides GPU information (assuming it is a NVIDIA gpu).
- `ncu --target-processes all -o <report name> <executable> [executable args]` will generate a compute report over all kernels given some executable and its args.

Notes

1. Benchmarks were run on a **NVIDIA GeForce RTX 4090** with **CUDA 12.2**.

2. The simulation was refactored to correctly run without rendering and over Linux. New logic was added to iterate until last particle velocity reaches a nonzero threshold ≈ 0, which is the point of "convergence". This is possible with a cudaMemcpy of `sizeof(uint32_t)`, which is assumed to be a very minimal cost per cycle, and an `atomicAdd` to a counter of "dead particles".
3. The original timestep (`#define DT_SIMULATION (1.0f / 20.0f)`) was updated to `(1 / 60.0f)` or once every second. That is a pretty smooth DT value.
    i. Regarding just how quickly the simulation "converges" even for `n=1` - that is a subjective rate based on my modeling, it does render - the base code was stripped of rendering capability.
4. Thread size and particle size vary and are tested as hyperparameters. Block size is according to the following:

```
size_t problem_sz = g_particles.h_maxParticles;
g_blocksPerGrid = dim3((problem_sz + g_threadsPerBlock.x - 1) / g_threadsPerBlock.x);
```

5. `test/...` is where many of the .sh scripts were made to iterate over various particle sizes and thread counts.
    i. The files are labeled in `cout/{particle ct}/`.
6. GPU Memory:
    i. Global Memory (~25 GB, of which only ~70 MB is storable in L2 cache)
        a. I am allocating `vec3 pos`, `vec3 vel`, and `float radius`, which is

$$2 * sizeof(vec3) + sizeof(float) = 28 \text{ bytes per particle}$$

        b. The **theoretical limit** to particle allocations on the GPU was just under **1 billion** particles:

$$\frac{\text{device memory}}{sizeof(\text{particle})} = \frac{25393692672 \text{ bytes}}{28 \text{ bytes}} \approx 906917595 \text{ particles}$$

        That said, out of concern for potential margin of error (and other users...), I made the upper bound `7,500,000` particles. My maximum particle runtime allocation was **20.4GB**.
        c. There are of course some other overhead allocations, i.e. `d_deadParticles` (isn't ideal for global, but can't be in `__constant__`) either.
    ii. Constant Memory (~64 KB, cached in L1)
        a. I allocated 4 `__constant__` members due to their constant nature, and frequent or "hot" use across all particles for collision checks and whatever else. These were `size_t d_maxParticles`, `size_t d_numPlanes`, `vec3 d_planeP[6]`, and `vec3 d_planeN[6]`.
        b. These may seem small, but GPU constant space is very limited, around about **64 KB**.
7. NCU Concerns

i. It is clear that while global accesses are necessary in this case, compute reports "**60%** excessive sectors" - less memory is used than is transferred. For example look at this function:

```
__device__ void solveConstraints(int idx, const glm::vec3* pos, const glm::vec3* vel, const
                                 glm::vec3& x_new, glm::vec3& v_new, float& dt, const glm::vec3
// Avoid at rest particles otherwise our counter will be inaccurate
if (glm::length(v_new) < STOP_VELOCITY) {
    return;
}

// Plane Collisions //
for (int i = 0; i < d_numPlanes; i++) {
    const glm::vec3& p(d_planeP[i]), n(d_planeN[i]);
    const glm::vec3& x(pos[idx]), v(vel[idx]); // FIXME: UNCOALESCED GLOBAL ACCESS (UNNECES

    glm::vec3 new_p = p + (radii[idx] * n);
    ...
}
```

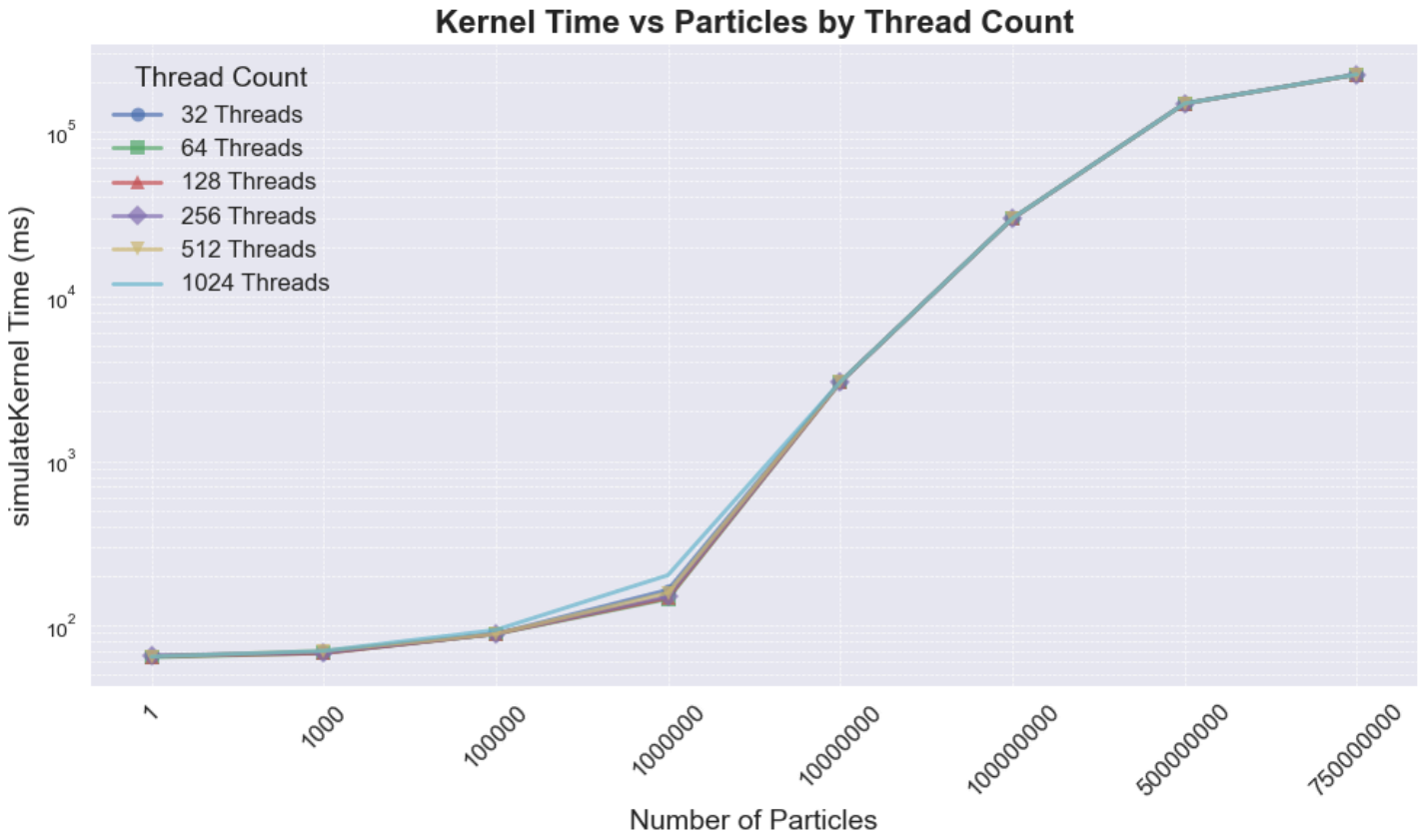
- It is not necessary to make a retrieval from global memory 6 times per plane for the same position.
- Even though we only ask for 12 bytes of our float3, memory boundaries will chunk global memory into sectors, which are usually ~32 bytes per chunk. The data is then retrieved in sectors, meaning when we ask for pos[0], we are perhaps getting pos[1], and **some** of pos[2].
- The reason **some** of pos[2] is problematic is that we now have to do an entirely additional fetch for just 4 bytes of pos[2]. Therefore, it is ideal to align your "object" into even chunks of the sector size.

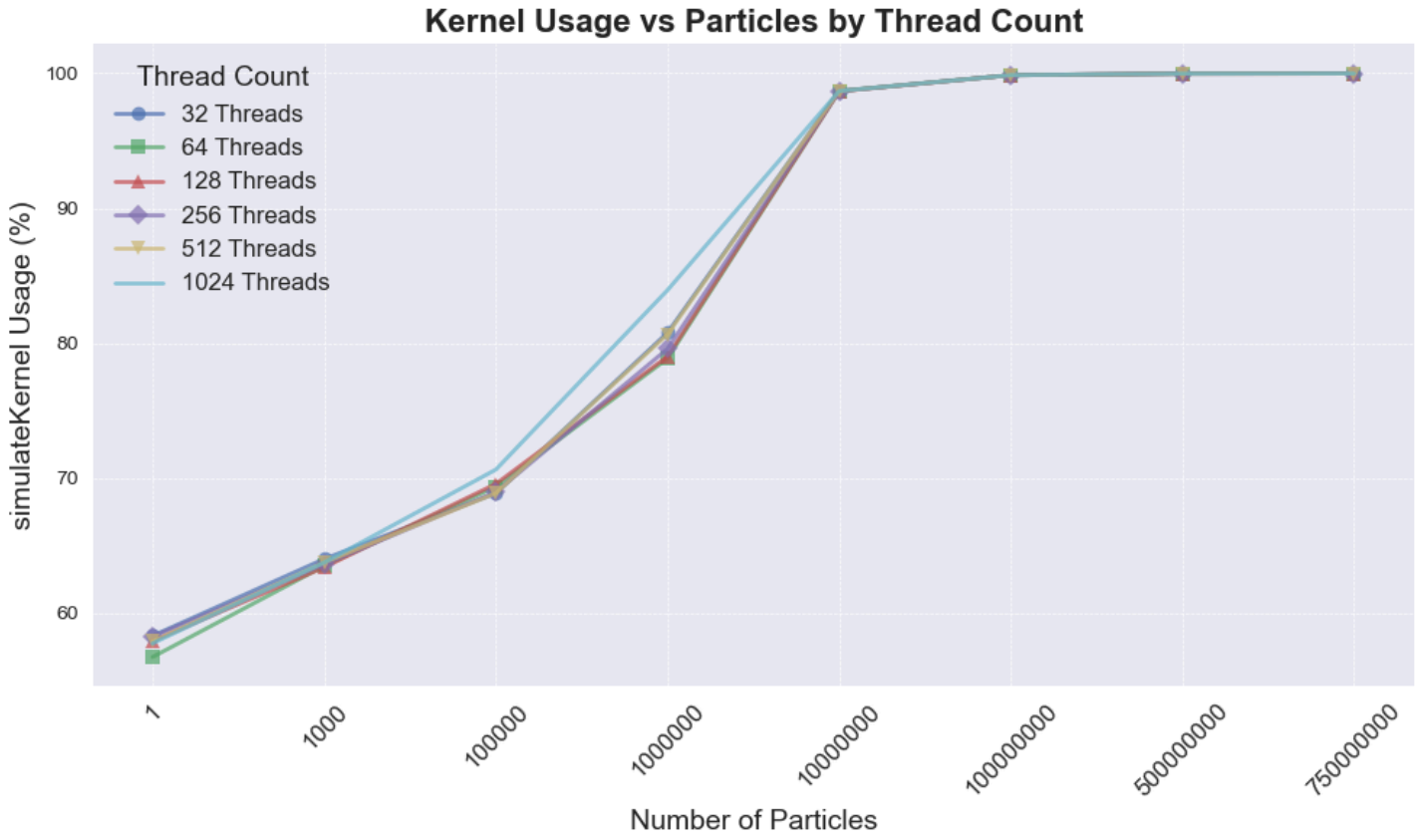
Improvements

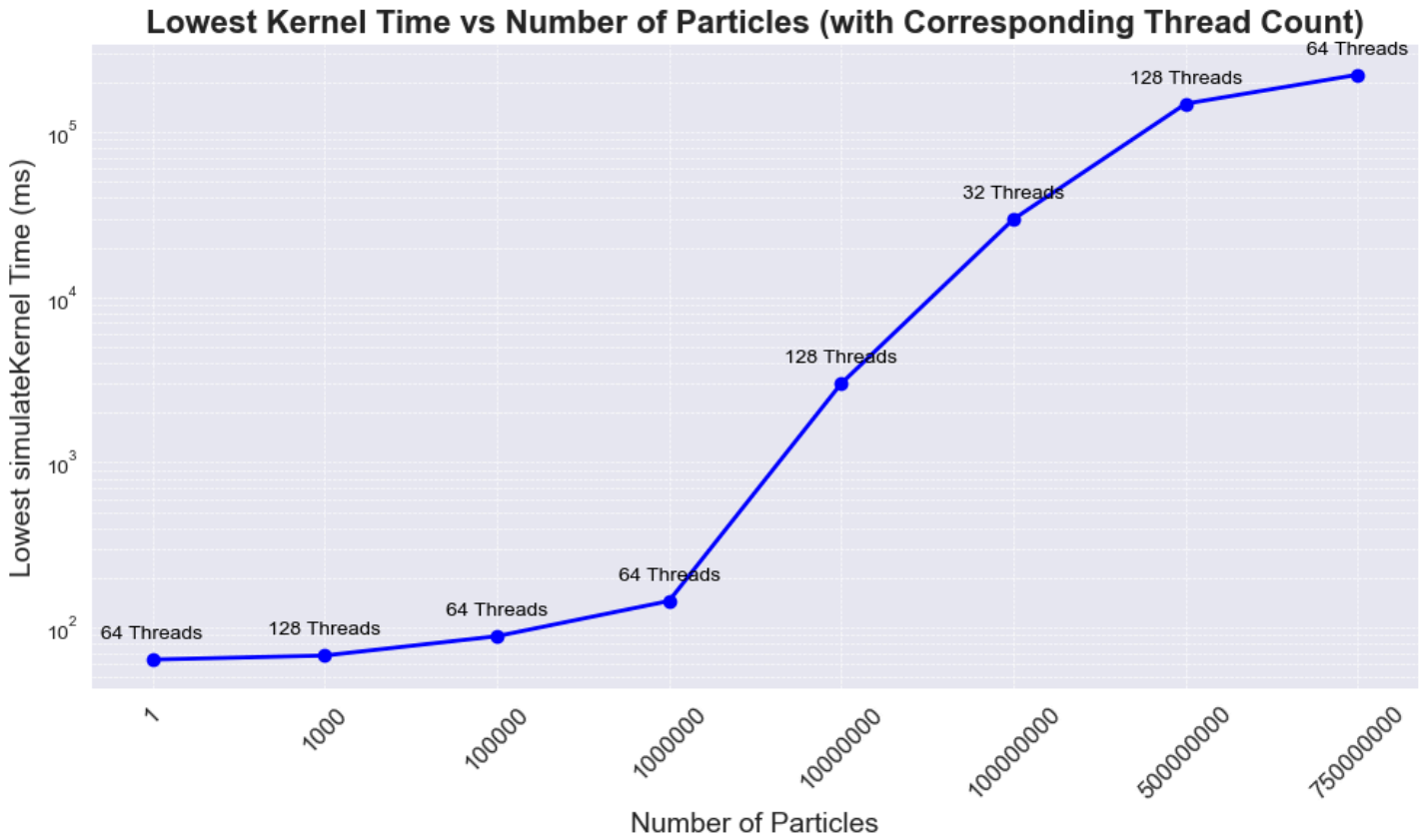
1. Padding was added, moving `vec3` to `vec4` for both pos and vel. This update brought $sizeof(\text{particle}) = 2 * sizeof(\text{vec4}) + sizeof(\text{float}) = 28$.
    i. This was done due to the uncoalesced global memory access, which was a major bottleneck. The idea was that sector size, at 32 bytes, is better stored with two vec4s, meaning no request would ever need to use another fetch request or cycle to get the necessary memory.
2. Any redundant fetches from global memory were refactored
    i. `const glm::vec3& x(pos[idx]), v(vel[idx]);` was moved outside of the loop iterating 6 times per `solveConstraints` iteration. This doesn't sound like a lot, but when it happens 5 times too many for each particle, it scales quickly.

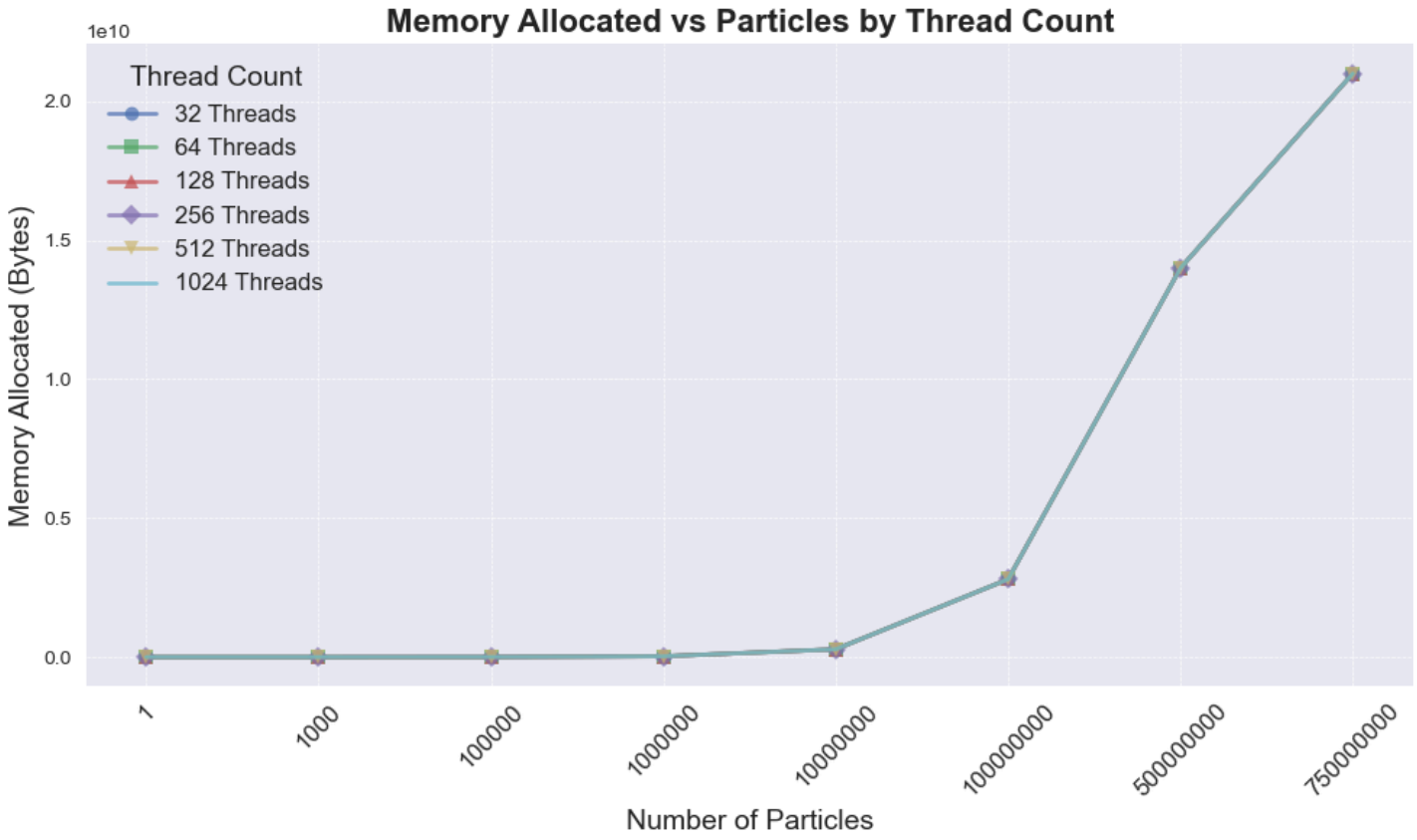
3. For the most part, the results of this were nonexistent. I would like to run more conclusive tests.

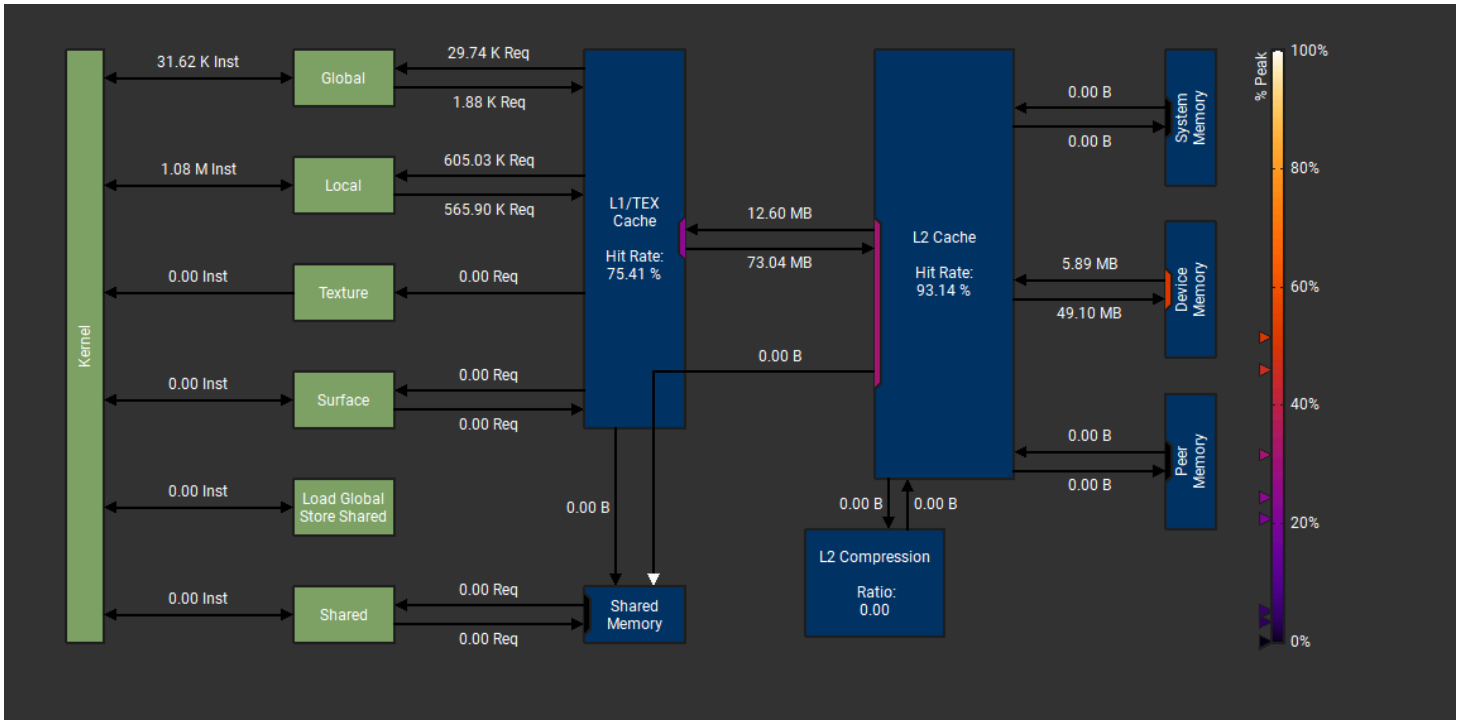
Plots & Images

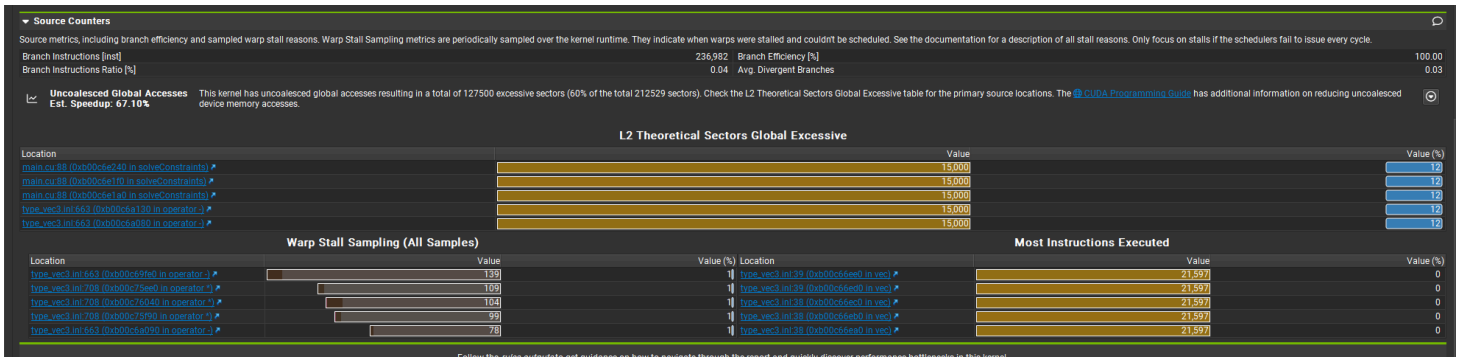



Kernel Usage vs Particles by Thread Count
Thread Count
32 Threads
64 Threads
128 Threads
256 Threads
512 Threads
1024 Threads
simulateKernel Usage (%)
100
90
80
70
60
1
1000
100000
1000000
10000000
100000000
500000000
750000000
Number of Particles


Lowest Kernel Time vs Number of Particles (with Corresponding Thread Count)
Lowest simulateKernel Time (ms)
10^5
10^4
10^3
10^2
64 Threads
128 Threads
64 Threads
64 Threads
128 Threads
32 Threads
128 Threads
64 Threads
1
1000
100000
1000000
10000000
100000000
500000000
750000000
Number of Particles


Memory Allocated vs Particles by Thread Count
1e10
Thread Count
32 Threads
64 Threads
128 Threads
256 Threads
512 Threads
1024 Threads
Memory Allocated (Bytes)
2.0
1.5
1.0
0.5
0.0
1
1000
100000
1000000
10000000
100000000
500000000
750000000
Number of Particles


Kernel
31.62 K Inst
Global
29.74 K Req
1.88 K Req
1.08 M Inst
Local
605.03 K Req
565.90 K Req
0.00 Inst
Texture
0.00 Req
0.00 Inst
Surface
0.00 Req
0.00 Req
0.00 Inst
Load Global Store Shared
0.00 Inst
Shared
0.00 Req
0.00 Req
L1/TEX Cache
Hit Rate: 75.41 %
12.60 MB
73.04 MB
0.00 B
0.00 B
Shared Memory
L2 Cache
Hit Rate: 93.14 %
0.00 B
0.00 B
0.00 B
0.00 B
L2 Compression
Ratio: 0.00
0.00 B
0.00 B
System Memory
5.89 MB
49.10 MB
Device Memory
0.00 B
0.00 B
Peer Memory
% Peak
100%
80%
60%
40%
20%
0%



Source Counters

Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

Branch Instructions [inst]	236,982	Branch Efficiency [%]	100.00
Branch Instructions Ratio [%]	0.04	Avg. Divergent Branches	0.03

Uncoalesced Global Accesses
Est. Speedup: 67.10%

This kernel has uncoalesced global accesses resulting in a total of 127500 excessive sectors (60% of the total 212529 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide has additional information on reducing uncoalesced device memory accesses.

L2 Theoretical Sectors Global Excessive

Location	Value	Value (%)
main.cu:88 (0xb00c6e240 in solveConstraints)	15,000	12
main.cu:88 (0xb00c6e1f0 in solveConstraints)	15,000	12
main.cu:88 (0xb00c6e1a0 in solveConstraints)	15,000	12
type_vec3.inl:663 (0xb00c6a130 in operator-)	15,000	12
type_vec3.inl:663 (0xb00c6a080 in operator-)	15,000	12

Warp Stall Sampling (All Samples)

Location	Value	Value (%)
type_vec3.inl:663 (0xb00c69fe0 in operator-)	139	1
type_vec3.inl:708 (0xb00c75ee0 in operator.*)	109	1
type_vec3.inl:708 (0xb00c76040 in operator.*)	104	1
type_vec3.inl:708 (0xb00c75f90 in operator.*)	99	1
type_vec3.inl:663 (0xb00c6a090 in operator-)	78	1

Most Instructions Executed

Location	Value	Value (%)
type_vec3.inl:39 (0xb00c66ee0 in vec)	21,597	0
type_vec3.inl:39 (0xb00c66ed0 in vec)	21,597	0
type_vec3.inl:38 (0xb00c66ec0 in vec)	21,597	0
type_vec3.inl:38 (0xb00c66eb0 in vec)	21,597	0
type_vec3.inl:38 (0xb00c66ea0 in vec)	21,597	0