15-418 Final Project: Position Based Fluid Simulation

Konwoo Kim *konwook@andrew.cmu.edu*

Lawrence Chen *ldc2@andrew.cmu.edu*

Summary

We implement a sequential and parallel position-based fluid simulator in C++ and CUDA and a renderer in OpenGL. We analyze the performance of our implementations across multiple scenes and achieve speedups of up to 30x when run on a Windows machine with an AMD Ryzen 9 5900HS CPU and an NVIDIA GeForce RTX 3060 Laptop GPU. We provide a live demo of real-time simulation and rendering at 144 frames per second/10 updates per second for 10,000 particles. Our demos and code can be found at the project website.

1 Background

1.1 Problem Description

We're interested in the problem of efficient simulation of incompressible fluids e.g. fluids with constant density. Existing approaches to this problem like smoothed particle hydrodynamics correctly enforce incompressibility but are sensitive to density changes and require small time steps. Instead, we adopt the approach of position-based fluids from Macklin & Müller (2013). Position-based fluids attempt to solve the problem using a position based dynamics framework which is more stable and allows for larger time steps as well as real-time applications.

The main idea of position-based fluids is that given an arbitrary particle and its neighbors, we iteratively enforce the constraint that the particle density at its position stays near a constant resting density (the definition of an incompressible fluid). If we enforce this for all particles and reach convergence, the density at all positions will be constant as desired.

1.2 Formalizing Position-Based Fluids

Formally, given a rest density ρ_0 and n particles where particle i has position $\mathbf{p}_i$, mass m_i, and density ρ_i, the ith constraint takes the form:

$$C_i(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n) = \frac{\rho_i}{\rho_0} - 1 = 0$$

We enforce this constraint for all $1 \leq i \leq n$ where we compute the density of particle i using a smoothed particle hydrodynamics density estimator of the following form:

$$\rho_i = \sum_{j \in N(i)} m_j W(\mathbf{p}_i - \mathbf{p}_j, h).$$

The estimator is taken over the set of all neighbors j of particle i, denoted by $N(i)$, where W can be an arbitrary kernel with radius h. We follow Macklin & Müller (2013) and use the Poly6 kernel for density estimation and the Spiky kernel for gradients.

In order to enforce this constraint, every iteration, we solve a system of equations where we have one constraint per particle. If we denote the positions of all n particles as $\mathbf{p}$, we wish to solve for $\Delta\mathbf{p}$ so that the new positions of the particles are closer to satisfying all of the constraints:

$$C_i(\mathbf{p} + \Delta\mathbf{p}) = 0 \text{ for all } 1 \leq i \leq n \iff C(\mathbf{p} + \Delta\mathbf{p}) = 0 \text{ where } C \text{ is a matrix of all constraints}$$

We approximate $\Delta\mathbf{p}$ using the constraint gradient and perform Newton steps to update the particle positions:

$$\Delta\mathbf{p} \approx \nabla C(\mathbf{p})\lambda \implies C(\mathbf{p}+\Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla C^T \nabla C \lambda = 0$$

Using this approximation, for each particle i, we can derive closed-form solutions for λ_i and $\Delta\mathbf{p}_i$:

$$\lambda_i = \frac{-C_i(\mathbf{p})}{\sum_{k\in N(i)} |\nabla_{\mathbf{p}_k} C_i|^2 + \epsilon} \text{ and } \Delta\mathbf{p}_i = \frac{1}{\rho_0} \sum_{j\in N(i)} (\lambda_i + \lambda_j) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$$

where ϵ is a relaxation parameter for regularization. We can iteratively update the particle positions in this way until convergence.

1.3 Techniques for Realistic Simulations

In addition to the baseline version of the above approach, we implement several add-ons as described by Macklin & Müller (2013) to produce more realistic simulations.

1.3.1 Tensile Instability

To avoid particle clustering when a particle has few neighbors, we add a corrective pressure term following (Monaghan, 2000):

$$s_{corr} = -k \left(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta\mathbf{q}, h)} \right)^n$$

where $n, k, \Delta\mathbf{q}$ are hyperparameters. The position update for $\mathbf{p}_i$ changes to include this term as follows:

$$\Delta\mathbf{p}_i = \frac{1}{\rho_0} \sum_{j\in N(i)} (\lambda_i + \lambda_j + s_{corr}) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$$

1.3.2 Vorticity Confinement

Because position-based methods use damping on the updates of particle velocities, vorticity confinement can replace the lost energy. We compute the vorticity for particle i using the curl:

$$\omega_i = \nabla \times \mathbf{v} = \sum_{j\in N(i)} (\mathbf{v}_j - \mathbf{v}_i) \times \nabla_{\mathbf{p}_j} W(\mathbf{p}_i - \mathbf{p}_j, h)$$

Then, we compute a corrective force which we can apply to the particle velocities (Macklin & Müller, 2013):

$$\mathbf{v}_i = \mathbf{v}_i + \epsilon(\mathbf{N} \times \omega_i) \text{ where } \mathbf{N} = \frac{\eta}{|\eta|} \text{ and } \eta = \nabla|\omega|_i$$

We implemented this as one of our project goals but chose to leave it out of our final simulator due to unstable behavior.

1.3.3 XSPH Viscosity

To better model coherent motion, we use XSPH viscosity which updates the velocity with an artificial viscosity term (Schechter & Bridson, 2012):

$$\mathbf{v}_i = \mathbf{v}_i + c \sum_{j\in N(i)} \mathbf{v}_{ij} W(\mathbf{p}_i - \mathbf{p}_j, h)$$

where c is a hyperparamter.

1.4 Implementing Position-Based Fluids and Rendering Sequentially

Now, we describe the core data structures and operations we use in our sequential implementation of position-based fluids and renderer. We implemented everything from scratch.

The core parts of our implementation are contained in several disjoint C++ modules: `Renderer`, `Simulator`, and `main`. Figure 1 shows how these layers interact with each other within the overall system while the pseudocode for `main` shows the `Renderer` and `Simulator` interface.

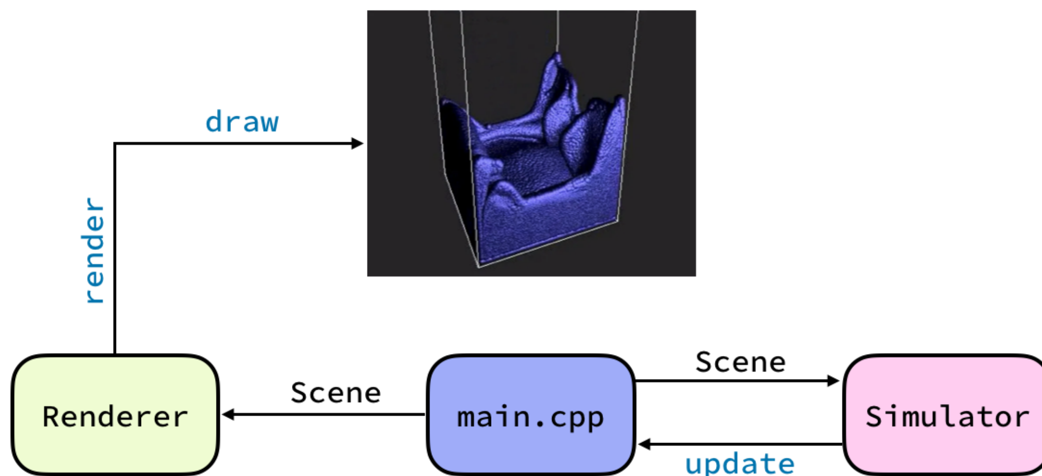


Figure 1: Fluid simulation system diagram.

Algorithm 1 main

```
1: Load scene from file
2: Initiate Simulator
3: Start up rendering thread
4: for iteration = 1, 2, ... do
5:     Simulator->update(scene)
6:     Tell rendering thread that a new scene is ready
7: end for
```

1.4.1 Sequential Simulation

We designed a generic `Simulator` class so that we could easily extend it with a `SequentialSimulator` or a `ParallelSimulator`. In either case, the `update` method will take in a `Scene` as input which contains a list of `Particles` where each `Particle` has state containing its position, velocity, and mass. As output, `update` will modify the positions of the `Particles` in place such that they satisfy the position-based constraint update. The pseudocode for this is shown below.

Algorithm 2 update

```
1: Initialize particles and compute initial forces
2: Recompute grid and neighbors
3: for iteration = 1, 2 do
4:     Compute densities
5:     Compute lambdas
6:     Compute delta positions
7:     Update positions and collisions
8: end for
9: Compute velocities and positions
10: Post-process (XSPH, Vorticity)
11: Update velocities
12: Update Particles in input Scene
```

As our pseudocode for `update` shows, `SequentialSimulator` contains several data structures to complete the position-based constraint update.

We have a `grid` data structure which assigns particles to a uniform grid where every dimension of a cell has value h so that we can query for neighboring particles in sub-quadratic complexity. We represent this as a continguous array which we index into using two separate arrays: `grid_sizes` which contains the number of neighbors of each particle and `grid_offsets` which contains the offsets from the beginning of `grid`.

To compute the position-based constraints, our `SequentialSimulator` also contains three arrays to represent the `densities` (ρ), `lambdas` (λ), and `delta positions` ($\Delta\mathbf{p}$) which store their respective values as the position updates are computed.

1.4.2 Sequential Rendering

Our `Renderer` implementation receives a `Scene` as input and visualizes it. We implemented this for debugging purposes and our end goal of real-time position-based fluid simulation.

At a high-level, `Renderer->draw` will process the input scene, clear the canvas, draw a particle bounding box, and draw the individual particles. For user inputs, we allow the user to look around using the mouse, move horizontally using WASD, and move up/down using the spacebar and CTRL keys, respectively.

The bounding box is drawn with 12 white lines, while each particle is drawn as a blue sphere with Phong shading. We implemented vertex and fragment shaders for the bounding box and particles to visualize them.

Since we needed to be able to render hundreds of thousands of particles quickly, we used instanced rendering. This is a technique that allows us to reduce the amount of information that needs to be sent from the CPU to the GPU, all in a single draw call. Using this, we only needed to transfer a single position for each particle, rather than every single vertex on each spherical mesh.

1.5 Parallelization

The sequential implementation of `update` is highly parallelizable in several ways. Each major step in update involves an operation which must be computed over all particles. Because this problem is highly data-parallel and we're interested in simulating hundreds of thousands of particles, using CUDA threads is an approriate choice for parallelization.

Thus, each operation like computing the densities of all particles can be done in parallel using a CUDA kernel which should provide speedup which scales with the number of particles. One sequential dependency is the computation each particle must perform over its neighbors. This is because we can't launch nested kernels and it would be infeasible to assign the work of the total number of neighbors over all particles to a single kernel call. It also wouldn't provide much benefit as there is already good enough work balancing in dividing up the particles.

The spatial locality of the sequential version can be improved by maintaining a more intelligent `grid` data structure. Because the density and gradient kernel computations are over neighbors, we want neighboring particles in our grid to be close to each other in memory. We can enforce this by sorting the particles in parallel relative to their grid cell indices. We discuss this in further detail in the following section.

Our problem is not amenable to SIMD execution because the main speedup we get is from the data parallelism on the GPU and not execution on the CPU.

Finally, we achieve better real-time performance by implementing rendering in a separate thread so that our frame rate is not capped by our update rate.

2 Approach

Next, we discuss our parallel implementation of position-based fluids in CUDA, our multi-threaded renderer, and our iterative process of optimization. Like our sequential implementation, we implemented everything from scratch.

2.1 Tech Stack

We used C++ for the base code and sequential simulator, CUDA for the parallel simulator, GLFW and GLAD libraries for OpenGL rendering, the GLM library for vector/matrix calculations, and CMAKE to build our project. We also had Python scripts for benchmarking and scene building.

The code was benchmarked on a Windows machine with an AMD Ryzen 9 5900HS CPU and an NVIDIA GeForce RTX 3060 Laptop GPU.

2.2 Implementing Parallel Position-Based Fluids and Rendering

2.2.1 Parallel Simulator

Our `ParallelSimulator` follows a similar interface to the `SequentialSimulator`, so the behavior follows Figure 1 and the pseudocode in Algorithm 2 describes what happens in each `update` call as well. Unlike the `SequentialSimulator`, the `ParallelSimulator`, performs each line in parallel using a kernel function. The class is structured to call these kernels with wrapper host functions.

For all of the kernel functions, we map the work over all particles using the same pattern: each thread within each block is responsible for doing the appropriate computation for a single particle. As an example, we'll walk through `compute_densities`, which is responsible for recomputing the density at each particle.

To begin, the host function wrapper makes a call to the corresponding kernel:

```
compute_densities_kernel<<<_blocks, _threads>>>(particles, neighbors,
                                                densities, neighbor_starts,
                                                neighbor_sizes, _n);
```

Within the kernel call, each thread begins by calculating `idx`, which is the index of the particle it's responsible for. If this is not a valid index, it returns immediately.

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx >= n) return;
```

Finally, it does the computation for the particle it's responsible for. In this case, it iterates through all neighbors and accumulates their contribution to the density using the appropriate kernel and stores the result in a global array.

```
Particle &p = particles[idx];
double density = 0.0;
for (int ni = neighbor_starts[idx]; ni < neighbor_starts[idx] + neighbor_sizes[idx]; ni++) {
```

```
    int neighbor_id = neighbors[ni];
    Particle &neighbor = particles[neighbor_id];
    density += GC.mass * poly6(p.new_pos - neighbor.new_pos, GC.h);
}
densities[idx] = density;
```

We make some changes from the `SequentialSimulator` to the `ParallelSimualtor` to better map to a parallel machine.

Our `SequentialSimulator` uses the `Constants` namespace to access hyperparameters and constants needed in our implementation, but the `ParallelSimulator` can't use this within device and kernel functions. Instead, we initialize a `GlobalConstants` struct on device constant memory using the `__constant__` qualifier. Using this, we can refer to constants through global, static memory accessible by all threads.

Additionally, we change our grid and neighbor computation to have better spatial locality while also being parallelized. To achieve better spatial locality, we aim to maintain a global array of particles such that neighboring particles are written in nearby memory. We follow the work of Green (2008) for fast fixed-radius neighbor computation which shows that using parallel counting sort results in a 4x reduction in the number of kernel calls per frame versus parallel radix sort.

First, we present our parallel implementation of grid recomputation below.

```
__host__ void ParallelSimulator::recompute_grid() {
    compute_bins<<<_blocks, _threads>>>(particles, bins, _n);
    thrust::exclusive_scan(thrust::device, bins, bins + _total_cells, prefix_bins);
    compute_sorted_grid<<<_blocks, _threads>>>(particles, prefix_bins, grid, _n);
    compute_grid_starts<<<_blocks, _threads>>>(particles, grid_starts, prefix_bins, _n);
}
```

In order to efficiently apply parallel counting sort, we introduce a `bins` data structure. This is a global array with size equal to the total number of cells in the grid. The `compute_bins` kernel will populate it so that the entry in bin i equals how many particles are in the ith grid cell. We implement this using `atomicAdd` so that we can still take advantage of data parallelism.

Once we've computed `bins`, we can efficiently populate `prefix_bins` in parallel using the thrust implementation of exclusive scan. If there are m particles in bin i, these will be written to memory starting from index i to index $i + m$ of `prefix_bins`. We can compute the sorted grid in this manner in parallel by once again using `atomicAdd` within the `computed_sorted_grid` kernel to atomically increment the index of `prefix_bins` each time a particle is written to the grid. The implementation of this kernel is shown below.

Finally, we compute the `grid_starts` for every cell in parallel so that we can efficiently find neighbors in our future kernels.

2.2.2 Multi-threaded Rendering

Originally, we ran our renderer sequentially within our main loop, where we draw a frame right after each physics update. However, as we tested higher and higher particle counts, this design became problematic because the frame rate would be capped by update rate. This made moving and looking around much harder, as we would need to wait for the next physics update to see how the perspective had changed.

Our renderer now runs on a separate thread that is spawned by the main program. Its job is to draw frames as fast as possible while polling for new scenes. The pseudocode for this is shown in Algorithm 3. The check for whether a new scene is available is done using an atomic boolean that can be accessed by both the main thread and the rendering thread. Copying of the new scene is done atomically using a mutex lock.

This new design allows us to move and look around smoothly, even if the physics updates are coming in at a slower pace.

Algorithm 3 Rendering thread

1: Initiate `current_scene`
2: **while** true **do**
3:     **if** new scene is available **then**
4:         Atomically copy new scene into `current_scene`
5:     **end if**
6:     draw(`current_scene`)
7: **end while**

2.3 Optimization

Our `ParallelSimulator` underwent several iterations of optimization and debugging. We discuss some of the main improvements we found as well as failure cases and unimplemented alternatives.

2.3.1 Conditional Evaluation of Gradients

When computing the values of λ_i in `update`, we have to compute the gradient of the constraint values. However, the closed-form solution of this gradient involves a conditional as shown below from Macklin & Müller (2013):

$$\nabla_{\mathbf{p}_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_{j \in N(i)} \nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k = i \\ -\nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k \neq i \end{cases} \tag{1}$$

Our kernel for computing the values of λ_i was directly calling a device function containing this conditional. Each particle would iterate through neighbors and call the function, which meant the slow $k = i$ case would occur for a neighbor, but it could be any arbitrary one. This meant that particles in the thread group who took the fast branch would still have to wait for other particles who took the slow one.

To optimize this, we split our gradient computation function into two separate functions: a slow one and a fast one according to the conditional. We then compute the slow function for all the particles at the same time and do the fast function afterwards, thus improving work balancing. This provided an average speedup of 1.5x over our previous `ParallelSimulator`.

2.3.2 Physical Kernel Computations

From profiling, we discovered that almost all of the time per update was spent in evaluating our `poly6` and `gradSpiky` functions. These are physical kernels used for density estimation and constraint computation, hence they are called frequently. Our first iteration of `poly6` is shown below:

```
__device__ double poly6(const dvec3& r, const double h) {
    double r_mag = length(r);
    if (r_mag > h) return 0.0;
    return 315.0 / (64 * GC.pi * pow(h, 9)) * pow(h * h - r_mag * r_mag, 3);
}
```

We made three main optimizations to this, which in total resulted in around a 4x speedup increase. First, `pow` is slow because it is a generic function which raises a float to a float power. We can speed it up by abusing the fact that the exponents are integers and using multiplication instead. Second, we inlined the function using `__forceinline__`. Finally, the if statement results in unnecessary branching, so we replaced it with a ternary operator. The final `poly6` code is shown below:

```
__device__ __forceinline__ double poly6(const dvec3& r, const double h) {
    double r_mag = length(r);
    double powh3 = h * h * h;
    double powh9 = powh3 * powh3 * powh3;
    double h_rmag = h * h - r_mag * r_mag;
```

```
    double h_rmag3 = h_rmag * h_rmag * h_rmag;
    return r_mag > h ? 0.0 : 315.0 / (64 * GC.pi * powh9) * h_rmag3;
}
```

We made very similar optimizations to the `gradSpiky` function.

2.3.3 Thread Block Sizing

We grid-searched different values to use for the number of threads per block and found that using 32 threads per block gave the best speedup.

2.3.4 Failure Cases and Alternatives

We'll first look at some specific optimizations we tried which failed, and then discuss some higher level design decisions we thought about or didn't finish.

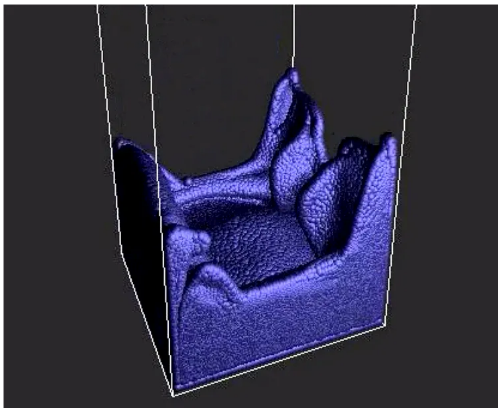
For optimizing kernel computations, we looked at caching the coefficients containing constants and computations of h on device constant memory and rewriting operations using CUDA math intrinsics. We were surprised that caching the coefficients didn't provided a negligible difference in performance as we were already reading from device constant memory for the value of π, but we didn't ablate this in depth.

Some additional optimizations we were planning on implementing but didn't finish include restructuring our kernels to use the `__ldg` intrinsic for read-only data to reduce the latency of global memory accesses and merging several of our simulation kernels to compute the neighboring particles only once. For the latter optimization, as we discuss in our results, we found that the primary overhead was kernel computation time so it's unclear how much it would have helped.

Some higher level design decisions we thought about were the tradeoffs of using separate arrays for storing computed data like $\Delta\mathbf{p}$ and λ versus storing this information within the `Particle` structures and implementing all of our pointers using `thrust::device_ptr`. For the former, we believe that it would have provided us with better spatial locality, and we decided against the latter for simplicity.

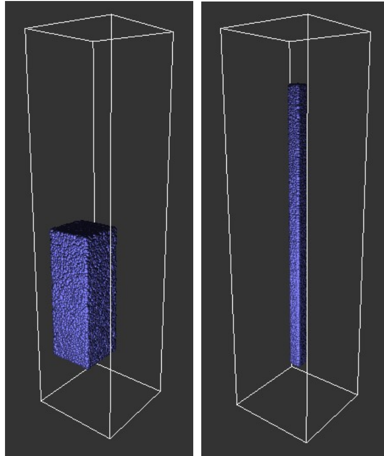
3 Results

We were successful in achieving what we planned as we achieved all of the baseline goals from our milestone report. As a brief overview of some major results, we successfully achieved real-time simulation and rendering of 10,000 particles, supported simulations up to 100,000 particles, and achieved speedups of up to 30x compared to our sequential implementation. We believe our choice of parallelizing on a GPU was sound.



3.1 Experimental Setup

We tested our code on two types of scenes, “blob” and “stream”, which have different initial particle distributions.



For each scene type, we varied the number of particles using powers of 2 from 1024 to 131072. The sequential simulator runs out of memory for 131072 particles, so we didn't collect any data for that case.

For performance, we measured the average time for calls to `Simulator->update(scene)` to return, as well as the average breakdown of these times into smaller steps. We averaged these measurements over 500 iterations per simulation. For real-time rendering, we additionally compute the number of frames rendered per second.

We use our single-threaded CPU-based `SequentialSimulator` as our baseline and compare it against our CUDA implementation of `ParallelSimulator`.

3.2 Speedup Results and Analysis

Figure 2 shows graphs of average times for a complete `update` call to finish.

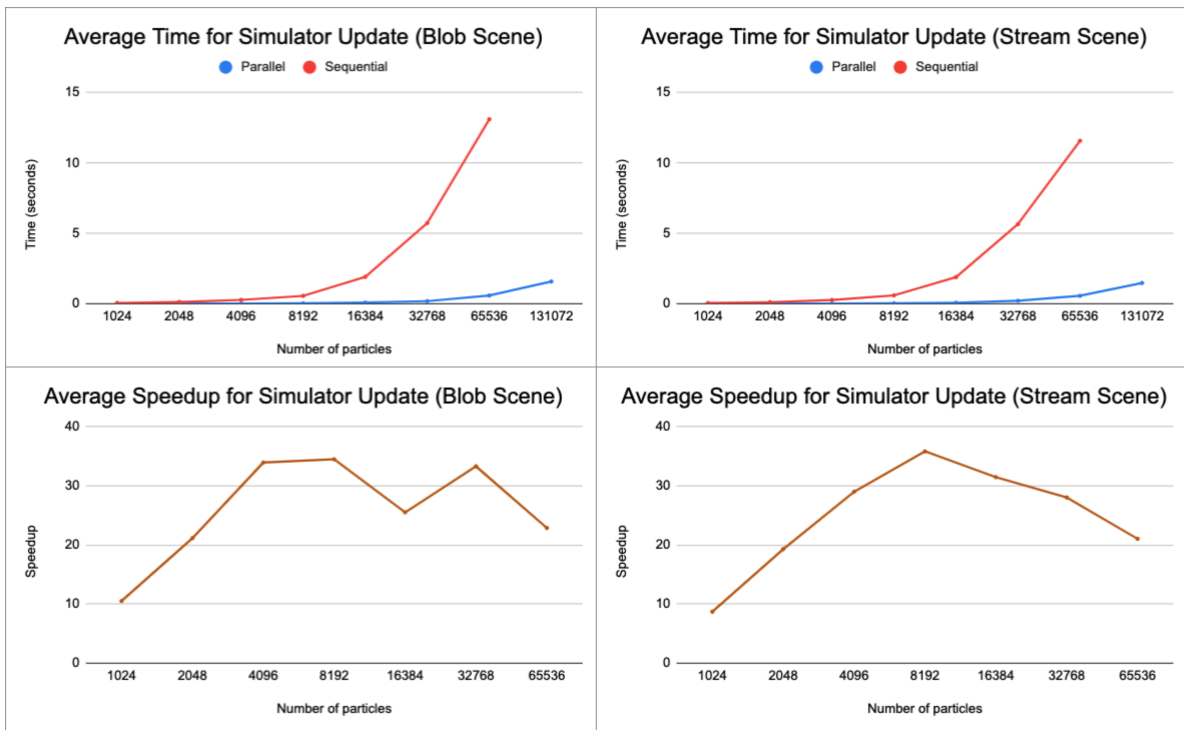


Figure 2: Speedup results across both types of scenes and simulators.

We observe that the parallel simulator is able to achieve speedups of 10x-20x at low particle counts, and 20-30x at high particle counts.

Although we'd hope to see speedup increase as a function of the number of particles, there are two areas that tend to have lower speedup: at the lowest number of particles and at the highest number of particles. There are three explanations for this phenomenon.

First, running CUDA code has a dependency of extra setup due to memory movement, which impacts simulations with low particle counts more. This is supported by the higher proportion of time spent in the "Initialization" component in the graphs in the following section. Figure 5 shows that for the lowest number of particles, initialization takes up 20% of the total update time, while for the highest particle sizes, it's less than 1%. This explains why speedup initially increases.

Furthermore, higher particle counts have lower relative speedups because we average the update times over 500 iterations of simulation, but convergence happens earlier within the first 200 iterations. Despite the different initial distribution of particles, the average updates in Figure 2 and breakdown of time in Figures 3-5 across scenes are nearly identical. This supports our claim as the steady state of an incompressible fluid will result in similar update times across the scenes. Since Figure 2 shows the relative speedup of the parallel implementation in the steady state is lower, we would expect better speedup if we averaged our results over a lower number of initial iterations.

Finally, having more particles results in fundamentally different simulation behavior. At low particle counts, the particles are spread out thinly along the bottom of the volume, resulting in less particle interaction overall. However, at high particle counts, there is room for multiple layers of particles to stack up, resulting in a higher number of neighbors and more interactions. This has a negative impact on the parallel renderer because there will be more variance in computation time per particle (this is speculation; we did not gather data for this), which results in more idle time. This same effect does not happen for the sequential renderer.

3.3 Deeper Analysis

We divide our update into several smaller components and time each component.

- Initialization: Update particle velocities based on gravity. For the parallel renderer, this also includes copying the particles into device memory.
- Gridding/neighbors: Calculate contents of each grid cell and compute neighbors for each particle.
- Densities: Compute density value at each particle location.
- Lambdas: Compute lambda value for each particle.
- Deltas: Compute delta value for each particle.
- Position/velocity updates: Update each particle's position and clamp them inside the bounding box.
- Post-processing: Perform XSPH viscosity. For the parallel renderer, this also includes copying the particles back to host memory.

The results for the "stream" scene were very similar, so we show them in a smaller size in Figure 5. This is the case because we average the update times over 500 iterations of simulation, but convergence happens earlier.

We observe that on average, more than 70% of time is spent in calculating lambdas and deltas. These functions iterate through all neighbors of each particle and accumulate values using the poly6 and gradSpiky kernels. This explains why optimizing the implementations of these two functions resulted in such drastic speedups. We profiled our code by replacing the physical kernel computations with no-ops to verify that the limit on speedup was from the kernels and not the neighbor computation, which is only at most 5% of total time.

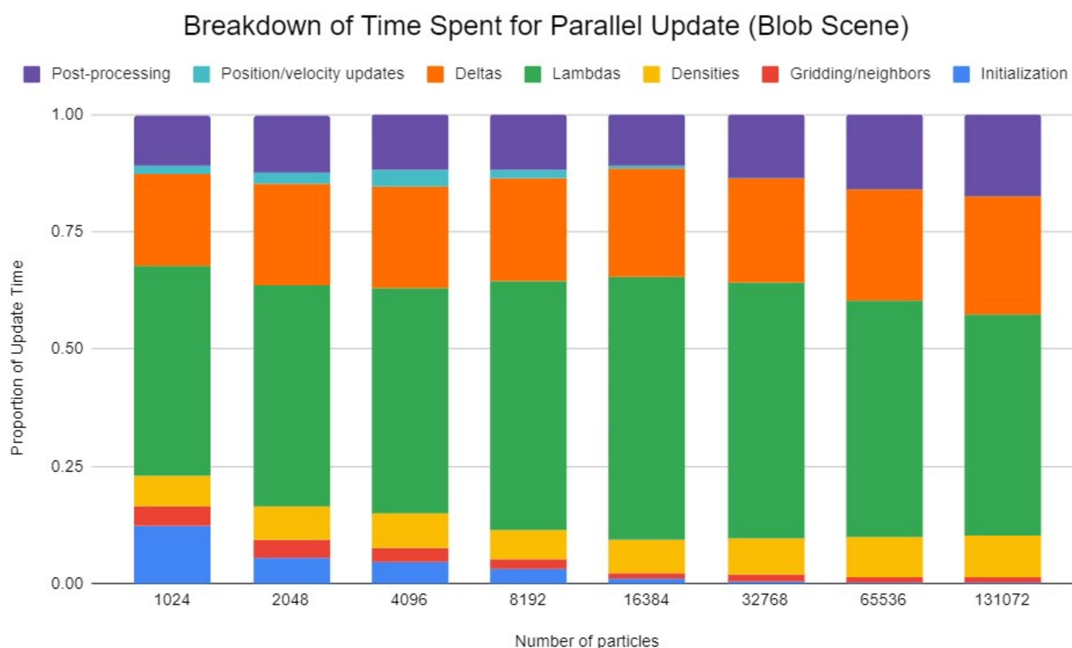


Figure 3: Breakdown of parallel update for blob scene into distinct components.

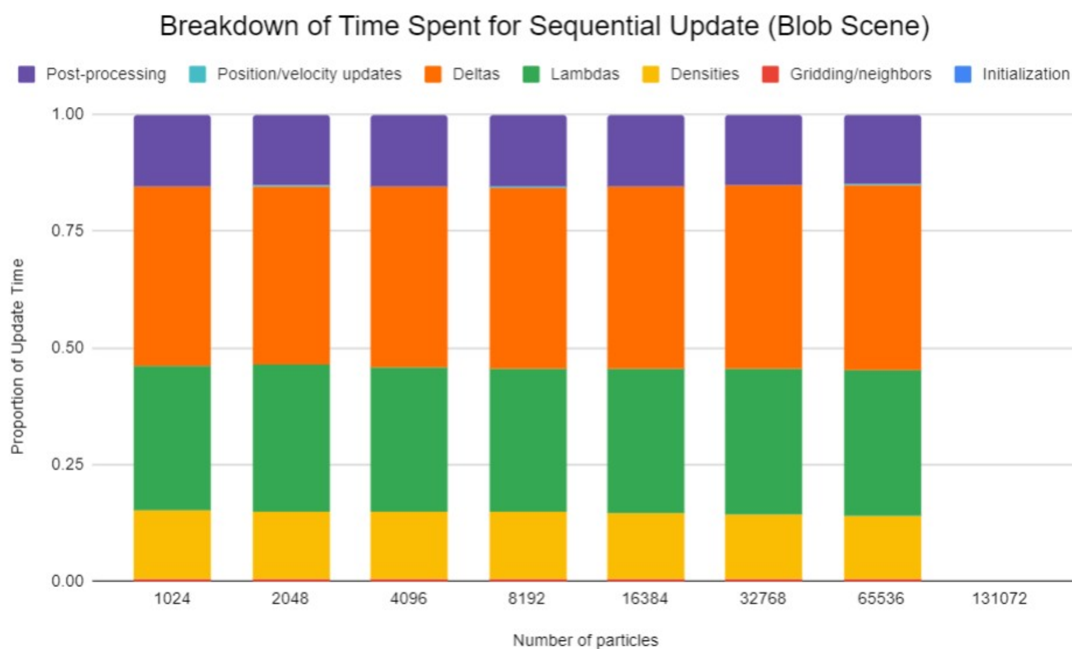


Figure 4: Breakdown of sequential update for blob scene into distinct components.

Additionally, there is only one device function in our parallel code that requires synchronization, which is the atomic add needed for particle gridding. The relatively low amount of time spent in “Gridding/neighbors” suggests that this isn’t a bottleneck and that our code isn’t memory-bound from reads or writes to global memory.

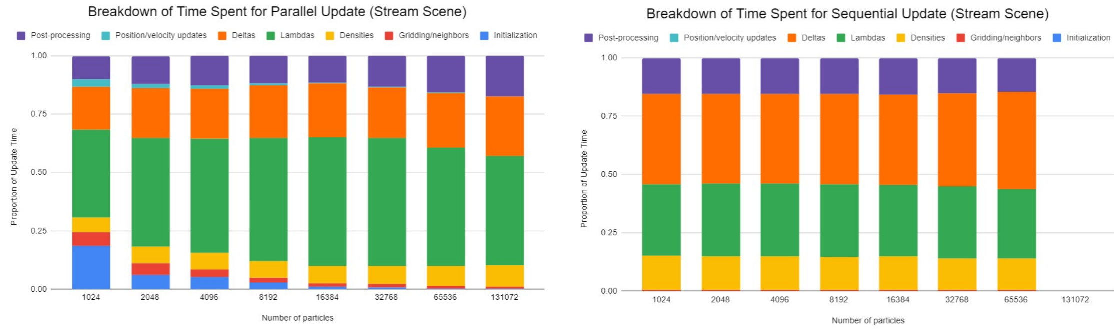


Figure 5: Breakdown of parallel/sequential update for stream scene into distinct components.

Additionally, we see that the distribution of work is much more uniform across different numbers of particles for the sequential simulator than for the parallel one. This makes sense because there is a lot of setup needed to run CUDA code, which becomes less significant the as the quantity of work increases. The sequential code does not need these extra setup steps. At high particle counts, the parallel work distribution becomes closer to the sequential one.

3.4 Real-time Rendering

Using the parallel simulator, we are able to simulate 10,000 particles in real time at a comfortable 10 updates per second, with 144 frames per second. The sequential simulator can only achieve 0.5 updates per second, although it can also render 144 frames per second.

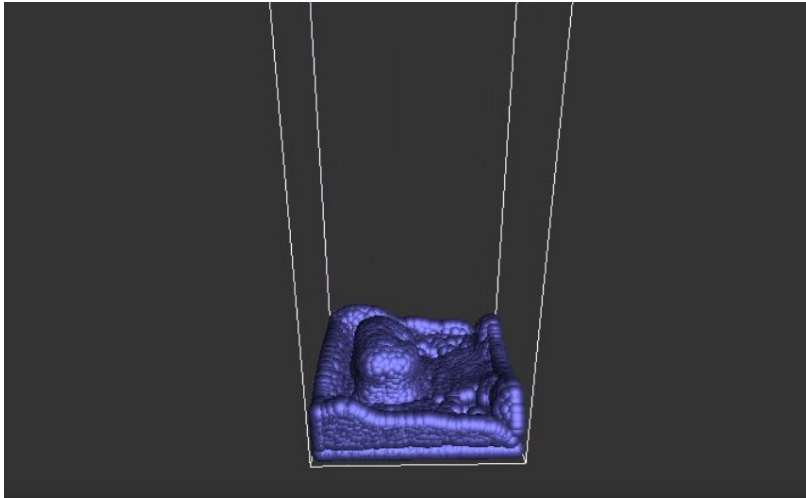


Figure 6: Real-time rendering with 10k particles.

4 Division of Work

The division of work was split equally between Konwoo and Lawrence as shown in Table 1.

Task	Konwoo	Lawrence
Renderer Implementation	✗	✓
Sequential Implementation	✓	✓
CUDA Implementation	✓	✗
Infrastructure and Scripting	✗	✓
Pair Debugging	✓	✓
Result and Demo Generation	✗	✓
Project Report	✓	✓
Project Poster	✓	✗

Table 1: Division of project work.

References

Simon Green. Cuda particles. *NVIDIA whitepaper*, 2(3.2):1, 2008.

Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4): 1–12, 2013.

Joseph J Monaghan. Sph without a tensile instability. *Journal of computational physics*, 159(2):290–311, 2000.

Hagit Schechter and Robert Bridson. Ghost sph for animating water. *ACM Transactions on Graphics (TOG)*, 31(4):1–8, 2012.