# Parallel Smoothed Particle Hydrodynamics Simulation
# 15418 Final Project Report

Haoying Zhang (haoyingz), Qilin Sun (qilins)

December 10, 2023

Github repo
Project website

## 1 Summary

In this project, we introduce an advanced parallel version of Smoothed Particle Hydrodynamics (SPH), comparing it to a sequential baseline. The sequential baseline runs on a Ryzen 5 5600G CPU, while the parallel version leverages the power of an RTX 3080 GPU. We perform comprehensive benchmarking on these platforms to demonstrate the efficiency gains. Notably, the GPU-optimized version achieves a remarkable 30 frames per second (fps) with a large particle count of 65,536, significantly surpassing the CPU baseline which only manages 10 fps with a much smaller count of 400 particles. A key finding of our work is the substantial performance improvement (up to 10x speedup) gained by integrating a GPU-based bitonic sort, compared to a more basic GPU implementation. This showcases the potential of GPU processing in enhancing computational performance in particle dynamics simulations.

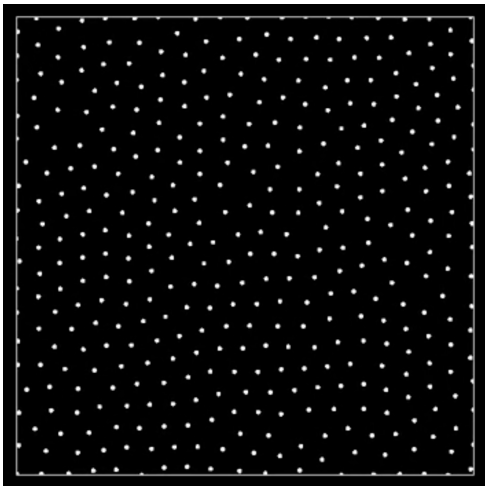

Figure 1: We implemented a particle-based fluid simulation

## 2 Background

The problem of fluid simulation could be approximated as a particle-level simulation problem, where each particle's state is updated by calculating interactions with neighboring particles, i.e., particles within the circles with radius $r$. The calculation of interactions includes multiple

inter-dependent steps. Figure 1 shows a frame from the simulation result. Section 2.1 provides the details of the algorithm.

### 2.1 Theory

This section is based off [1]. Each particle has a position $\mathbf{p}$ and velocity $\mathbf{v}$. To convert the discrete point mass distribution into a continuous distribution of mass, a smoothing function, or a kernel, is used. We define the kernel as

$$W(\mathbf{x}, \mathbf{p}) = \begin{cases} \frac{6}{\pi r^4}(r - \|\mathbf{x} - \mathbf{p}\|)^2 & \text{if } \|\mathbf{x} - \mathbf{p}\| < r \\ 0 & \text{o.w.} \end{cases}$$

where $r$ is the smoothing radius, $\mathbf{p}$ is the position of a particle, and $\mathbf{x}$ is a location we are interested in. Note that the term $\frac{6}{\pi r^4}$ normalizes the total mass of a smoothing kernel to 1, as the integral of $(r - x)^2$ over a circular region with radius $r$ is $\frac{\pi r^4}{6}$. The smoothing radius $r$ controls the region of influence of a particle; a location that is further than $r$ apart from a particle will not be influenced by it at all. The influence of a particle decays quadratically from its center location.

The density at a location is simply defined as the sum of the influence of all particles at that location, i.e.

$$\rho(\mathbf{x}) = \sum_i W(\mathbf{x}, \mathbf{p}_i)$$

where $i$ iterates through all particles. Now we introduce pressure,

$$P(\mathbf{x}) = \alpha(\rho - \rho_{des})$$

where $\rho_{des}$ is the desired density and $\alpha$ serves as a tunable gain that determines how aggressive the pressure is. Intuitively, pressure is proportional to how far the current density is from desired density. A location with high density will have a positive pressure; others will have lower or negative pressures. Note that this pressure is defined for each particle, in constrast to the weighted pressure defined below.

To query a property $A$, e.g. pressure, at a location, the following approximation is used:

$$\begin{aligned} A(\mathbf{x}) &= \int A(\mathbf{p}_i) W(\mathbf{x}, \mathbf{p}_i) dV \\ &= \int \frac{dm}{\rho(\mathbf{x})} A(\mathbf{p}_i) W(\mathbf{x}, \mathbf{p}_i) \\ &\approx \sum_i \frac{1}{\rho(\mathbf{x})} A(\mathbf{p}_i) W(\mathbf{x}, \mathbf{p}_i) \end{aligned}$$

where $m$ is taken as 1 to simplify the computation. This is reasonable as long as all particles have the identical mass.

Now, to query the weighted pressure, $\mathcal{P}$ at a point, we simply compute

$$\mathcal{P} = \sum_i \frac{1}{\rho(\mathbf{x})} P(\mathbf{p}_i) W(\mathbf{x}, \mathbf{p}_i)$$

That is, $\mathcal{P}$ is the weighted sum of pressures at many points. Now the central part of the simulation comes in. To make the particles move, there must be a force. The gradient of pressure is used to produce this force, and the acceleration on a particle is

$$\mathbf{a}(\mathbf{p}) = \frac{\nabla \mathcal{P}(\mathbf{p})}{\rho(\mathbf{p})}$$

By the chain rule,

$$\nabla \mathcal{P}(\mathbf{p}) = \sum_i \frac{1}{\rho(\mathbf{x})} P(\mathbf{p}_i) \nabla W(\mathbf{x}, \mathbf{p}_i)$$

The rest is simple: all needed to be done is update the positions and velocities accordingly, just like what happened in Assignment 3/4.

$$\mathbf{p}_{t+1} \leftarrow \mathbf{p}_t + \mathbf{v}_t \Delta t + \frac{1}{2} \mathbf{a}_t \Delta t^2$$
$$\mathbf{v}_{t+1} \leftarrow \mathbf{p}_t + \mathbf{a}_t \Delta t$$

However, when the above numerical integration is taken, the velocities of particle may blow up over time, eventually leading to diverging simulation. We suspect this is due to numerical integration error and switched to a second-order Runge-Kutta method. Let $\mathbf{s}_t$ be defined as the state of a particle at time $t$, i.e.

$$\mathbf{s} = \begin{bmatrix} \mathbf{p} \\ \mathbf{v} \end{bmatrix}$$

Then the Runge-Kutta method is defined as:

$$k_1 \leftarrow f(\mathbf{s}_t)$$
$$k_2 \leftarrow f(\mathbf{s}_t + \frac{2}{3} k_1 \Delta t)$$
$$\begin{bmatrix} \tilde{\mathbf{v}} \\ \tilde{\mathbf{a}} \end{bmatrix} \leftarrow \mathbf{s}_t + (\frac{1}{4} k_1 + \frac{3}{4} k_2) \Delta t$$
$$\mathbf{s}_{t+1} \leftarrow \begin{bmatrix} \mathbf{p}_t + \tilde{\mathbf{v}} \Delta t + \frac{1}{2} \tilde{\mathbf{a}} \Delta t^2 \\ \mathbf{v}_{t+1} + \tilde{\mathbf{v}} \Delta t \end{bmatrix}$$

where $f$ refers to the computation of $\mathbf{v}_t$ and $\mathbf{a}_t$, given the position of all particles. Because each particle has the property $\mathbf{v}$, it is taken for granted. The acceleration $\mathbf{a}_t$ is computed using density and the gradient of weighted pressure, as discussed above. Note that $f(\mathbf{s}_t + \frac{2}{3} k_1 \Delta t)$ is an expensive step, as it involves stepping the position and velocity of all particles by a delta of $\frac{2}{3} k_1 \Delta t$ and recomputing all densities, pressures, and pressure gradients.

After adopting this method, the simulation converges to a point where all particles exhibit little motion. We could have chosen the famous RK4 algorithm instead, but that will take much more time and is therefore unsuitable for our application.

## 2.2 Data Structures

**Core data structures:**
For a particle system simulator, we need to store the particles, and intermediate states. Therefore, we will have two particle buffers and a state derivative buffer. The reason we need two particle buffers instead of one is as follows:

In each iteration, our approach is not simply finding the acceleration of each particle and updating its position and velocity, but instead it computes the current acceleration **and** the acceleration after the particles have stepped a bit forward. This is where is second particle buffer comes to play: it is used to store the updated positions of each particle after stepping. In addition, the 2 accelerations are weighted and stored in a state derivative buffer. Finally, each particle's positions and velocities are updated according to the weighted acceleration.

To speed up the simulation, a simple idea would be dividing the grid into blocks with side length equal to the smoothing radius, as it reduces the number of particles that need to be looked up. The basic implementation will maintain a list of particles for each block, and in each iteration step the list is re-filled. To avoid synchronization overhead, every block will write to

its own buffer for the updated particles, iterate through all the particles, and store the relevant particles in the buffer.

Because iterating through the list is expensive, we want to find out a way to implement redistribution process that avoids this. Meanwhile, we want to keep the locality within the block - accessing all particles in a block should have good locality, as the basic implementation with dedicated list for each block.

# 3 Approach

Before we start, please acknowledge the purpose and scope of this project:

- This project mainly aims at exploring the ways of speeding up the simulation via parallel computing.
- The accuracy of the simulation and the theoretical foundation behind it are less relevant. The techniques presented in this work should be able to be adapted into another simulation methodology with relatively little effort.
- Visualization, which is the front-end of this project, is less important than the simulation process, which is the back-end. While we would deliver a demo in OpenGL, it would not be as sophisticated as one rendered from a CG software.
- Our implementation only concerns a 2D simulation context. It could be extended into 3D, but for the purpose of studying parallel computing, this goal is less important.

## 3.1 Steps of the simulation

In each iteration, there are 5 steps:

I. For each particle, compute the densities, pressures, and accelerations at the current time step.

II. For each particle, following the current velocity and acceleration, step forward for a fraction of a step, and compute the resulting positions and velocities.

III. For each particle, compute the new densities, pressures, and accelerations for the updated positions.

IV. For each particle, weigh and sum the 2 sets of accelerations.

V. Update the position and velocity of each particle. Clamp positions to be in the context and invert velocities as appropriate.

Note that step 2-5 each depend on the previous step, so no parallelism can happen between these steps.

An obvious way to optimize the runtime is to note that the smoothing kernel has a small region of influence $r$, so the simulation context can be subdivided into square blocks with side length $r$, as shown in Figure 2. This way, to compute densities and pressure gradients of particles inside a block, only this block and the 8 neighboring blocks need to be considered, which is exactly what happened in Assignment 4. In the next sections, we will explain how we utilize a fully data-parallel approach to speed up the simulation.

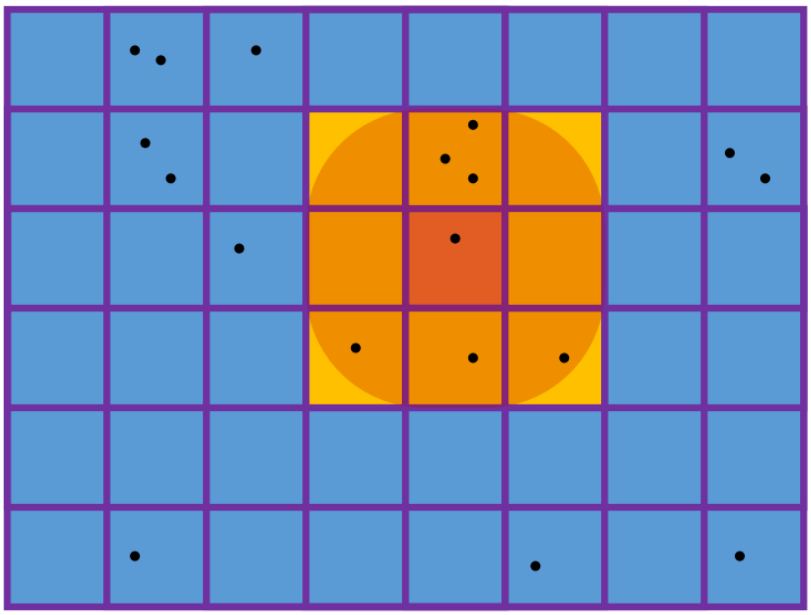

Figure 2: A block has 8 adjacent blocks that may contribute to the update of the block

### 3.2 Basic blocked implementation

A naive way to do this would be to partition all particles into blocks and only look up the relevant ones when dealing with a particle, but this will cost extra storage that inflates quickly as the simulation context size increases. In the basic implementation, each block maintains a list of particles that are in the block. As the number of particles in a block is unknown, each buffer needs to be allocated large memory to be able to contain all particles. Also, each block needs a thread which is responsible for iterating through the particle array and copying relevant particles into the buffer. This is done sequentially.

This could be done in parallel, though - firstly, we will find the indices of particles that belong to a particular block using the technique in homework 2, and copy them to their corresponding index in the buffer in parallel. However, we decided to move on to a fundamentally different implementation rather than spending more time on this baseline. After all, when the simulation context has size of $20 \times 20$ and the smoothing radius is 1, 400 blocks are needed, and the amount of space required by the blocks is 400 times the size of the particle array. This extreme wasteful use of space pushes us to consider an alternative.

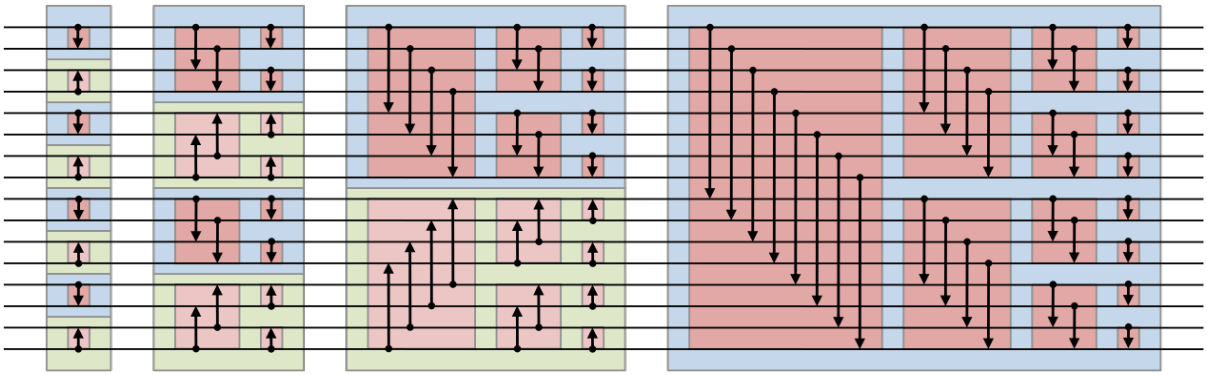

Figure 3: Bitonic Sort (From Wikipedia)

### 3.3 Optimized implementation with in-place sorting

Alternatively, we optimized our implementation by incorporating efficient in-place sorting, which both saves memory and improves locality. A key strategy to achieve high speedup is the bitonic sort [2]. We added in a field showing the block index in each particle `struct`. In each iteration, each particle's block index is computed using its latest position, and they are sorted by their block index. As we will demonstrate in the next section, our GPU implementation of bitonic sort is very efficient, as its run time is much shorter than that of computing the densities and pressure gradients - which is the inherent cost of the simulation itself. The bitonic sort employs a bufferfly network to achieve $O(\log^2 n)$ run time with $O(n)$ processes, where $n$ is the length of the array. Because GPUs are able to perform compare-and-swap with embarassing parallelism, bitonic sort is running especially fast. Figure 3 shows how exactly is sorting done.

### 3.4 Optimizing the number of threads for each particle

As of computing densities and pressure gradients, a straight-forward way to implement step I would be to create number of particles many threads in CUDA, each computing the influence of other particles on a particle. Yet we pay close attention to the presence of loops in individual threads and try to even them out among more threads as much as possible, and a seemingly better way to do this would be to let threads collaborate to process the 9 boxes each particle. However, when 2 threads are spawned for each particle, the first dealing with block 1-4 and the second 5-9, the performance worsens. At a particle count of $512 \times 512$ and initially the particles occupy 60% of the width and height of scene, the original method takes 58 ms on average while the other 67 ms. Creating more threads to collaborate leads to even worse performance, so we conclude that the most straight-forward method turns out to be the optimal one.

### 3.5 Benefits of implementation with in-place sorting

Using an in-place sorting algorithm has the following benefits:

1. We avoid the expensive memory copy operations.
2. We have better locality which can further reduces latency, as particles within a block are now adjacent in the particle buffer.

## 4 Results

We reiterate that computing acceleration twice is essential for the sanity of the simulation. If a naive integration method is used, the velocity of particles can blow up after a few time steps, resulting in total chaos in the simulation. Although a simpler approach can lead to a twofold speedup, we choose this more appropriate method. The run time distribution and relative speedups will be discussed in this section.

### 4.1 Benchmarking

The benchmark we are using is the run time per frame, in milliseconds. We do not use animation fps because the rendering process becomes the bottleneck when the particle gets large and fps becomes less informative about how much time is taken by the simulation itself. We will list the runtime of each of the 4 processes below:

1. Partitioning (either through copying into arrays or bitonic sort)
2. Compute densities and pressures

3. Compute pressure gradient
4. Compute acceleration
5. Update particles

Update particles refers to both the initial substepping and the final updating.

For benchmarking, all tests are run in a gravity-less context, with the desired density set to the total particle mass over the area of the context. This is better for understanding the performance of the simulation because particles will diffuse toward an even distribution over the context, so the comparison of performance is made fairer. We have 3 implementations for benchmarking:

- CPU sequential baseline implemented with blocks.
- GPU baseline implemented with blocks.
- GPU optimized version implemented with bitonic sort.

Each implementation is run with $32\times32, 64\times64, 128\times128, 256\times256$, and $512\times512$ particles. The particles initially occupy 80% of the width and height of the scene, i.e., if width is 10, the particles roughly occupy the x-range $[1, 9]$. The context has width 20 and height 20, and the smoothing radius of each particle is 1. The simulation is then run for 1000 steps, and the runtime of each step is measured in the host program and averaged. The following table 1 shows the total run time:

| Particle count | CPU-blocks runtime (ms) | GPU-blocks runtime (ms) | GPU-bitonic runtime (ms) |
|---|---|---|---|
| $32 \times 32$ | 8.82 | 1.39 | 1.71 |
| $64 \times 64$ | 68.83 | 4.19 | 2.28 |
| $128 \times 128$ | 1150.56 | 13.56 | 3.08 |
| $256 \times 256$ | 23018.86 | 84.61 | 7.55 |
| $512 \times 512$ | 364704.23 | 407.54 | 59.77 |

Table 1: Total runtime per iteration

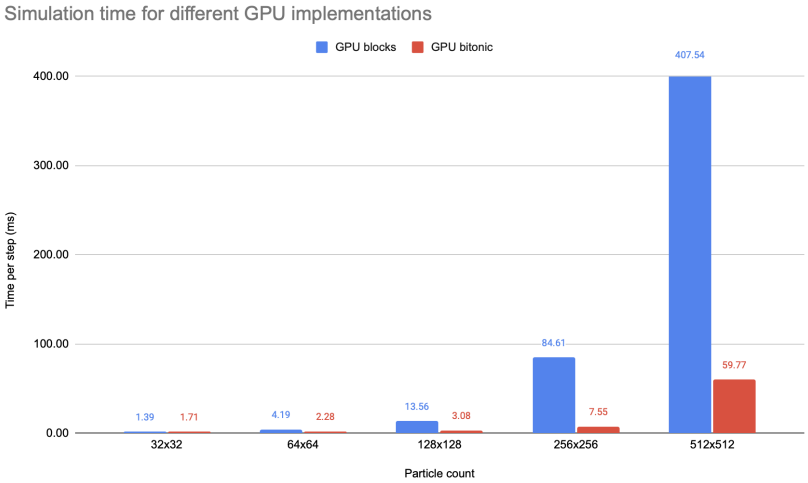


Figure 4: Compare simulation time for different GPU implementations

## 4.2 Analysis

Note that "Partition" refers to copying particles into arrays for each particle, for the CPU- and GPU-blocks versions, and bitonic sort for the GPU-bitonic sort version. When the number of particle is very large, the runtime for CPU runtime is very large. The main reason is that the CPU version is fully sequential, but even if perfect speedup can be achieved on Ryzen 5 5600G, which has 12 threads, the runtime for $512 \times 512$ particles would be 22794 ms, which is still much inferior to either GPU implementation.

Figure 4 shows the run time comparisons, while figure 5 shows the speedup of iteration runtime achieved by GPU-bitonic, compared to the GPU-blocks version. From this graph, it is clear that the bitonic sort implementation drastically outperforms the blocks version, with a maximum speedup of around 11 at $256 \times 256$ particles. We suspect $256 \times 256 = 65536$ threads to be the parallel capacity of the GPU.

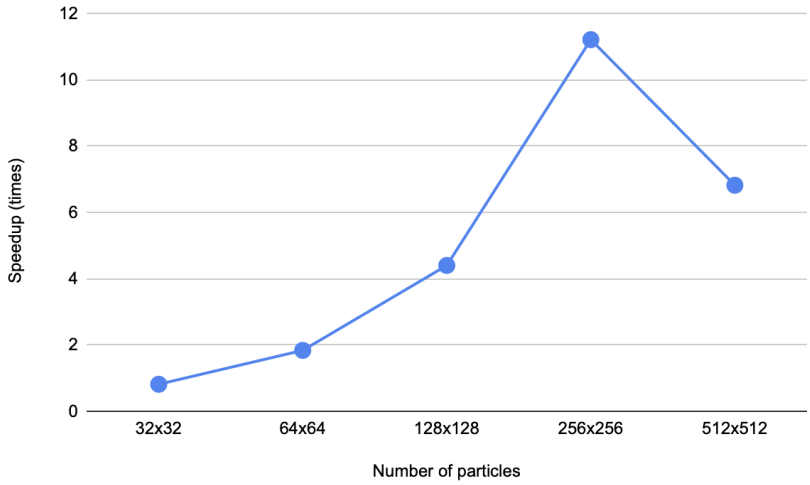


Figure 5: GPU-Bitonic Speedup over GPU-Blocks

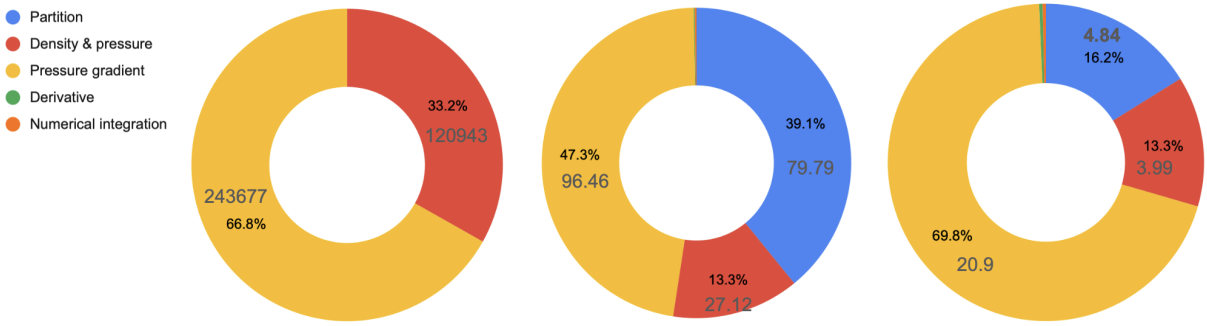


Figure 6: Runtime Breakdown for $512 \times 512$ particles (ms) - Left: CPU, Middle: GPU-Blocks, Right: GPU-Bitonic

Figure 6 shows run time breakdown for each implementation with particle count of $512 \times 512$. Observe that compared to GPU-blocks, GPU-bitonic spends a notably smaller portion of time partitioning the particles, i.e. performing bitonic sort. This means that the preparation for doing the most important work take less time, and the bulk of the time is spent on doing the actual work, which is desirable.

Also note that the arithmetic intensities of computing density, pressure, and pressure gradients are very high, because the attribute of a particle is only read once, followed by purely

arithmetic operations. This suggests that most of the time is actually spent on performing floating point math, which is the inherent part of the simulation.

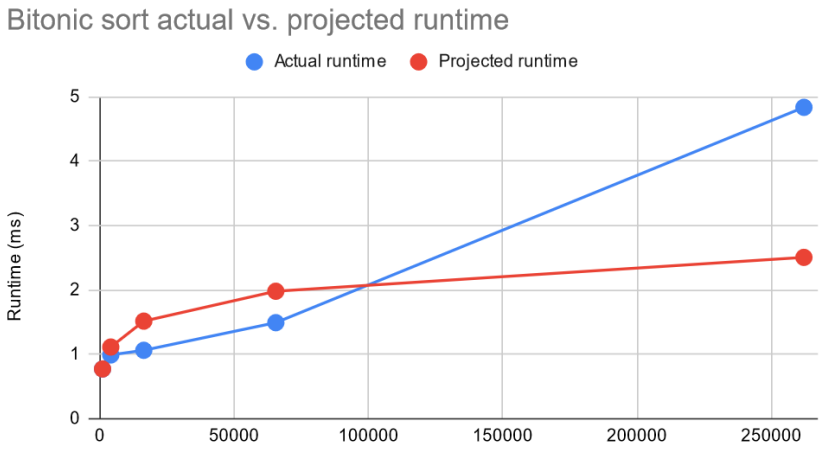


Figure 7: Bitonic sort analysis

In addition, we analyze how closely bitonic sort obeys its theoretical span. Figure 7 shows the actual runtime and projected runtime based on its $O(\log^2 n)$ span. Evidently, as particle count increases, the algorithm performs worse than the projected performance. This is expected, though, because the maximum number of threads on a GPU is limited, and exceeding this limit forces some threads to be executed sequentially, and the runtime therefore deviates from the ideal span.

### 4.3 Limitations

We now analyze how imbalanced particle distribution affects performance. With a greater desired density (a configuratble parameter in the simulation), particles will cluster together, thereby leading to imbalance in the number of particles per block. Hence, we performed a density sweep for a number of desired densities that are greater than the desired density. Table 2 3 shows the time taken for partitioning and time taken for computing densities, pressures, and pressure gradients. Data is collected in a context of height 10 and width 10 with $128 \times 128$ particle that initially occupy 80% of width and height. The simulation is run for 1000 steps, and the average time taken for selected densities is shown in the tables below 2 3. The speedup of bitonic version of blocks version is also reported.

| Density | GPU-bitonic | GPU-blocks | Speedup |
|---|---|---|---|
| 1 | 1.04544 | 4.39848 | 4.207300275 |
| 2 | 1.03254 | 4.49287 | 4.351279369 |
| 3 | 1.03409 | 4.51796 | 4.369020105 |
| 4 | 0.767291 | 3.52605 | 4.595453355 |
| 6 | 0.748081 | 3.31774 | 4.435001023 |
| 8 | 0.780222 | 3.34585 | 4.288330757 |
| 16 | 0.772616 | 3.47245 | 4.494406018 |

Table 2: Partition runtime vs. density

The complete data is graphed in 8. From the graph, it appears that the speedup is fairly stable across different densities. That is, the bitonic sort version only outperforms the blocks version by a constant. As the blocks get denser, its runtime is on the same order of the other version.

| Density | GPU-bitonic | GPU-blocks | Speedup |
|---|---|---|---|
| 1 | 1.421545 | 2.154864 | 1.51586056 |
| 2 | 2.756524 | 3.644236 | 1.322040367 |
| 3 | 4.083879 | 5.378436 | 1.316992007 |
| 4 | 4.69668 | 6.78355 | 1.44432876 |
| 6 | 2.847932 | 5.9134 | 2.076383846 |
| 8 | 8.47355 1 | 1.37198 | 1.342056163 |
| 16 | 5.90461 1 | 5.65927 | 2.652041371 |

Table 3: Computing densities, pressures, and pressure gradients runtime vs. density

We first analyze the speedup of bitonic sort over partitioning into blocks. The speedup being a constant means that the $O(\log^2 n)$ delay of bitonic sort degrades into the $O(n)$ delay of the other version. We speculate that it is a result of the GPU's parallel capacity being saturated as the number of particles reaches a threshold, as sorting $n$ particles require $\frac{n}{2}$ threads. When there are too many particles, it is no longer possible to perform all compare-and-swap in parallel. The performance therefore starts worsening.

We then analyze the speedup of computing densities, pressures, and pressure gradients. Recalling section 3.4, dedicating multiple threads for each particle in a balanced (average density) setting reduces performance, but in unbalanced situations this may actually improve the performance. That is, our approach only optimizes the runtime in balanced settings, but may not be the best for other settings. Hence, this task is left for future research, which may explore when to allocate more threads for each particle, perhaps dynamically, or even use a Quad-Tree data structure on the GPU. Our intuition for implementing the former is: the GPU passes the number of particles in each block to the host, and the host decides how many threads to create for each particle, depending on how dense its surrounding blocks are. We envision granularity control to be crucial for this purpose.

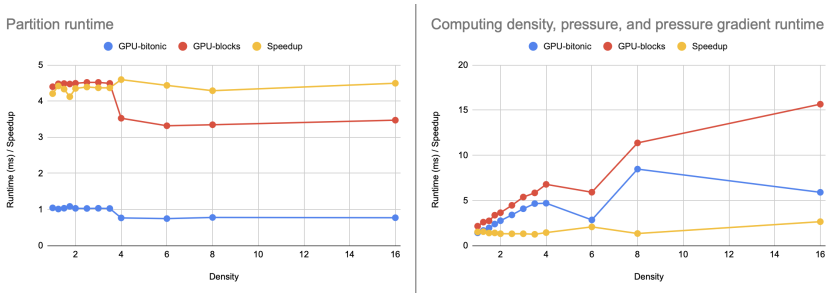


Figure 8: Bitonic sort analysis

# 5 Miscellaneous

## 5.1 Position clamping

Because the simulation context is a rectangular box with fixed size, if a particle is stepped out from the box, its position is clamped and velocity negated. Let $w$ be the width of the box, then the left boundary is $-\frac{w}{2}$ and the right $\frac{w}{2}$. Let the x-coordinate position and velocity of a particle be $p_x$ and $v_x$, respectively, then if $|p_x| \geq \frac{w}{2}$, the following is done:

$$p_x \leftarrow \begin{cases} \frac{w}{2} - (p_x - \frac{w}{2}) & \text{if } p_x \geq \frac{w}{2} \\ -\frac{w}{2} + (-\frac{w}{2} - p_x) & \text{if } p_x \leq -\frac{w}{2} \end{cases}$$

This is nothing but reflecting the coordinate of x across the leftmost or rightmost boundary. Note that this is not simply clamping, because the particle has travelled a little beyond the boundary and reflecting would be more accurate. The above math simplifies to

$$p_x \leftarrow \begin{cases} w - p_x & \text{if } p_x \geq \frac{w}{2} \\ -w - p_x & \text{if } p_x \leq -\frac{w}{2} \end{cases}$$

As for velocity, the math is much more straight-forward:

$$v_x \leftarrow -v_x$$

This assumes that all collisions are elastic and no energy is lost.

For the other direction(s), the clamping is done as similarly.

### 5.2 Power of 2 for bitonic sort

The bitonic sort by itself only runs on an array whose length is a power of 2. If the array is not exactly a power of 2, it has to be padded with dummy elements to reach a power of 2. For simplicity, we run the simulation only with some power of 2 number of particles. It is straight-forward to modify the program to accommodate other input lengths, but we deem that less important for the project and did not make it happen.

### 5.3 Computing pressure

The method used to compute pressure,

$$P(\mathbf{x}) = \alpha(\rho - \rho_{des})$$

is only accurate for gas. For liquid, however, it can be used as an approximation that is easy to compute. Since our project is not about accuracy, we deem it suitable to use this method instead.

## 6 Contribution

| | |
|---|---|
| Haoying Zhang | 40% |
| Qilin Sun | 60% |

Table 4: Contribution makeup

Qilin Sun did the background research and came up with the preliminary CPU sequential implementation (not the blocked version). Afterwards, Haoying Zhang and Qilin Sun collaborate to implement CPU blocked, GPU blocked, and GPU bitonic sort implementations. Data collection, analysis, and report writing, and poster making are all equally contributed by the group members.

## 7 Acknowledgements

We acknowledge the exciting YouTube video Coding Adventure: Simulating Fluids by Sebastian Lague for inspiring our initial ideation. We were able to quickly get into SPH and rapidly prototype our first version, thanks to this video.

We also acknowledge the courtesy of DROP Lab for offering us a platform with Ryzen 5 5600G CPU and RTX 3080 GPU. All of our development and experiments are run on those.

## References

[1] Peter J. Cossins. Smoothed particle hydrodynamics, 2010.

[2] Nassimi and Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C-28(1):2–7, 1979.