15-418 Final Project Report

Rohit Pillai (rrpillai), Anubhav Jaiswal (ajaiswal)

Summary

We implemented a parallel version of a fluid simulator to run on Nvidia GTX 1080 GPU's on the Gates machines, comparing it to an optimized sequential version on the CPU as a baseline. We also wrote a parallel CPU version using OpenMP to compare its results to the other two versions. We used the smoothed particles hydrodynamics (SPH) algorithm, which is just an approximation of the Navier Stokes equations in order to actually calculate the interaction between the different segments of the fluid. The word "particle" from now just refers to a small section of fluid.

The Navier Stokes equations are the equations we used to model the fluid as a continuum, and the SPH algorithm is used to simplify these equations. This algorithm has 2 main steps, which are:

1. Calculate the density of each portion of the fluid based on the other particles around it.
2. Calculate the acceleration of each particle, which is once again influenced by the particles around it.

Background

As mentioned in the previous section, we are implementing a parallel computational fluid dynamics (CFD) simulator for homogeneous and incompressible fluids that we can run on the GPU's in the Gates machines. In order to do this, we are using the SPH algorithm to approximate the Navier Stokes equations. These equations are displayed below and explained in the operations sections.

$$\frac{\partial u}{\partial t} = -(\mathbf{u} \cdot \nabla) u - \frac{1}{\rho} \nabla p + \nu \nabla^2 u + f_x,$$
$$\frac{\partial v}{\partial t} = -(\mathbf{u} \cdot \nabla) v - \frac{1}{\rho} \nabla p + \nu \nabla^2 v + f_y.$$

Eq. 1: Navier Stokes equation for velocity(x and y) of a particle

$$\rho_i = \frac{4m}{\pi h^8} \sum_{j \in N_i} (h^2 - r^2)^3.$$

Eq 2: SPH equation for density of a particle

$$\mathbf{a}_i = \frac{1}{\rho_i} \sum_{j \in N_i} \mathbf{f}_{ij}^{\text{interact}} + \mathbf{g},$$

Eq 3: SPH equation for acceleration of a particle

$$\mathbf{f}_{ij}^{\text{interact}} = \frac{m_j}{\pi h^4 \rho_j}(1 - q_{ij}) \left[15k(\rho_i + \rho_j - 2\rho_0) \frac{(1 - q_{ij})}{q_{ij}} \mathbf{r}_{ij} - 40\mu \mathbf{v}_{ij} \right],$$

Eq 3.1: Definition of $\mathbf{f}_{ij}$ from the previous equation

While the Navier Stokes equations assume every fluid to be a continuum (infinitesimally small particles), our implementation of the SPH algorithm allows us to discretize the fluids, representing small portions of the fluid as single particles. These particles contain information about the fluid they represent, including density, acceleration, position, mass, and velocities. All of these are with respect to the fluid within a particle. Thus we can model a fluid as many discrete particles interacting with each other, influencing only those particles that lie within a certain radius.

Key Data Structures

In order to represent all the features that were mentioned above, we decided to use a C struct called CurrState that held all the information to represent the current state of the fluid. We have arrays of size 2N (where N is the number of rendered particles) to represent the positions, velocities and accelerations of each particle, with the 2N term coming from the fact that these quantities are all vectors and thus contain an X and Y component. The density array is of size N since density is a scalar value so there is only one value per particle. The analogous struct in our CUDA implementation also contains pointers to arrays located on the device so we can perform computations using kernels, which read, write, and modify just those arrays.

Additionally, we had a C struct representing parameters that were essential to the simulation, such as the number of frames to be rendered, the number of steps per frame, the gravitational constant etc. Furthermore, we also stored static properties of the fluid such as the viscosity and size of each particle.

Operations on Data Structures

The most intensive operation performed on our data structures is computing the acceleration of each portion of the fluid, based on equation 3 as shown above. Here we see that the acceleration of each particle is dependent on its current density, the gravitational constant and the combined forces of interaction between it and all other particles(fij). However, since the density of a particle is not constant, it also needs to be computed via equation 2. From this equation, we see that each particle's density depends on its mass (m), its position (r) and its distance from surrounding particles (h). Since the density is inversely proportional to the distance between the particles, we can safely ignore all the particles that are far away from the current one being computed. We also have to calculate the force of interaction between all the particles and the current one, using equation 3.1.

Once we have calculated the acceleration and the density, we can update the velocities and positions of the particles. The velocity values for each particle aren't dependent on anything except the calculated acceleration for that particles, and the positions are dependent on the calculated velocities.

Breakdown of Operations

Currently, our algorithm only takes in a few parameters such as the number of frames we want to display, the number of steps to be taken per frame in the simulation, the elapsed time between each step we take, and the number of particles to be rendered on the screen. Given all these inputs, the program outputs a text file of particles' positions over time, that is later read from to actually visualize the simulation.

As mentioned before, updating the velocity of each particle is dependent just on the acceleration of the particle. Similarly, updating the position of the particle is dependent only on the velocity of that particle. Thus, we can easily parallelize these computations by having a kernel that updates each element in the array.

On the other hand, the density and acceleration of a particle are dependent on every other particle, which is very hard to parallelize. However, by avoiding an optimization trick we used in our serial version, we were able to parallelize it. Each step in our algorithm (i.e calculate accelerations, calculate density and update velocities and positions) is still sequentially dependent on one another, and we have no way of parallelizing that.

Approach

The main technologies we used for this project were C++ for most of the code in the serial and the parallel version and CUDA to write the code for the kernels in the parallel version**.** We also used OpenMP to run a parallel CPU version of the code. Additionally, we used OpenGL to render outputs of both the serial and parallel versions. All our code was being run on the GHC machines, and as a result, we used the NVIDIA GeForce GTX 1080 GPU's to run our CUDA code on.

Workload Mapping

With respect to parallelization in both the CUDA and OpenMP code, we launch as many threads as there are particles, with each thread performing work on a specific particle-a specific index in an array. However, we still iterate through the entire positions array in the calculateDensity kernel, and we iterate through the entire positions and density array in the calculateAccelerations array. We are unable to avoid this since each particle is dependent on every other particle in the fluid. Our CUDA code has the number of threads per block set to be 128 currently, since that seems to provide us with the most speedup for reasons mentioned later on. To minimize the overhead of repeatedly transferring memory from host to device and back, we copied all pertinent data over to the device in the beginning of the program and had the device perform all computation. Once finished we had to copy the results back over to the host in order to write these values to a file once per timestep for the visualizer to later use.

Difference between Serial and Parallel Algorithms

The mathematical foundation and base algorithm that we use in all versions are the same. First we update the densities of each particle, followed by its accelerations, velocities and positions. After these steps, we take into account any reflections on a particle due to hitting a boundary and output the final positions at that time into a file.

The only difference between our serial and parallel implementations is how we iterate through the arrays. In the serial implementation, we only iterate through our arrays (N ^2 - 1)/2, with N being the number of particles being rendered, since we take into account the fact that densities and accelerations are symmetric with respect to particles. Thus for the current particle, we only have to iterate through the particles that are after it in the array if we're going through it sequentially. This approach isn't possible with a parallel implementation however, since we would induce race conditions as multiple threads could be modifying the same position in the array. Thus, in the parallel case, we iterate through all the particles in the array for each particle.

Iterations of the program

Initially, we started off trying to do the fluid simulation using the Navier Stokes equation as is. However, while trying to find a way to actually solve the differential equation required for the Navier Stokes equation, we realized that it was a very hard problem and that using an approximation of the equation might be better. As a result, we decided to switch to the SPH algorithm, that approximates the Navier Stokes equations.

While we were implementing the Navier Stokes equations, we decided to represent our fluid as a 2-D grid, with each cell containing all the data pertinent to it. However, when we switched to our SPH algorithm, we realized that we were only reading a subset of all variables in any given function. Reading the entire struct just for one value is a poor use of cache locality as non-pertinent data will populate the cache from other fields of the struct. Thus, we switched the way we represent data to a struct containing one dimensional arrays for each given field to maximize cache locality.

We approached the CUDA implementation by copying over our serial implementation and trying to reduce dependencies. At first we did not have much difficulty writing the simpler kernels as there weren't really any data dependencies to be considered. However, we soon realized that by having half our computation on host and half our computation on device, we were incurring a large overhead of copying memory. To reduce this we knew we had to perform all the computation on the GPU, meaning all the code needed to be parallelized. This posed some challenges, however, because it meant we would need to remove all dependencies in our code so that everything could run on kernels. We approached this by restructuring the two sections of code that shared the most dependencies -- the calculations for density, and the calculations for acceleration. We made the tradeoff of increasing total work by a factor of 2, and decreasing span from O(N^2) to O(N) by having each thread iterate through all other particles to calculate its own acceleration and density values.

Starter Code

We used code from assignment 2 to start off our CUDA implementation and used the reference documents listed at the end of this report.

Results

Output

The below pictures show the initial configuration, an intermediate configuration and the final configuration of the fluid with the fluid divided into roughly 1500 particles.

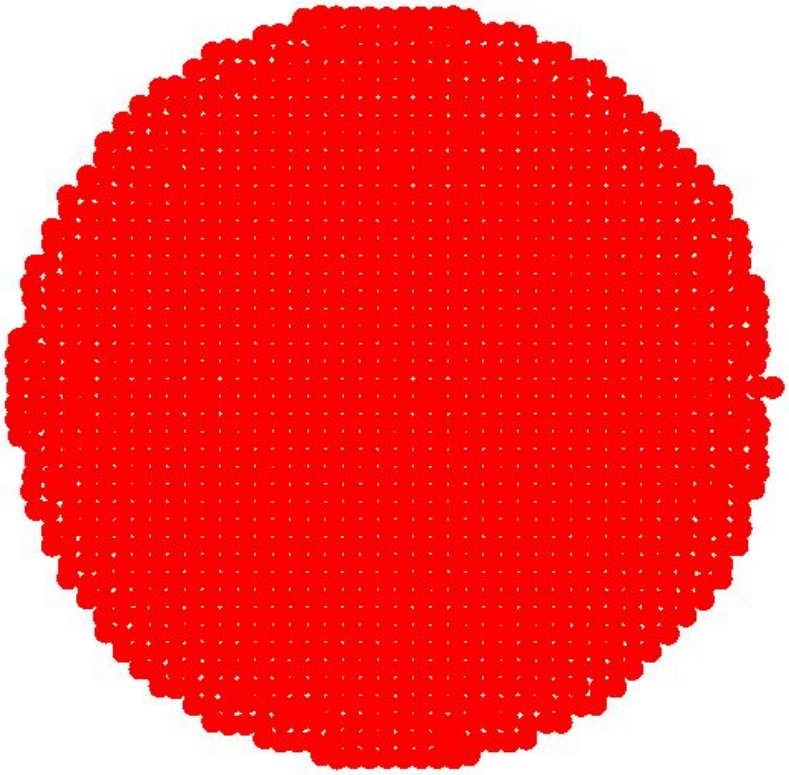


Fig 1. Initial configuration

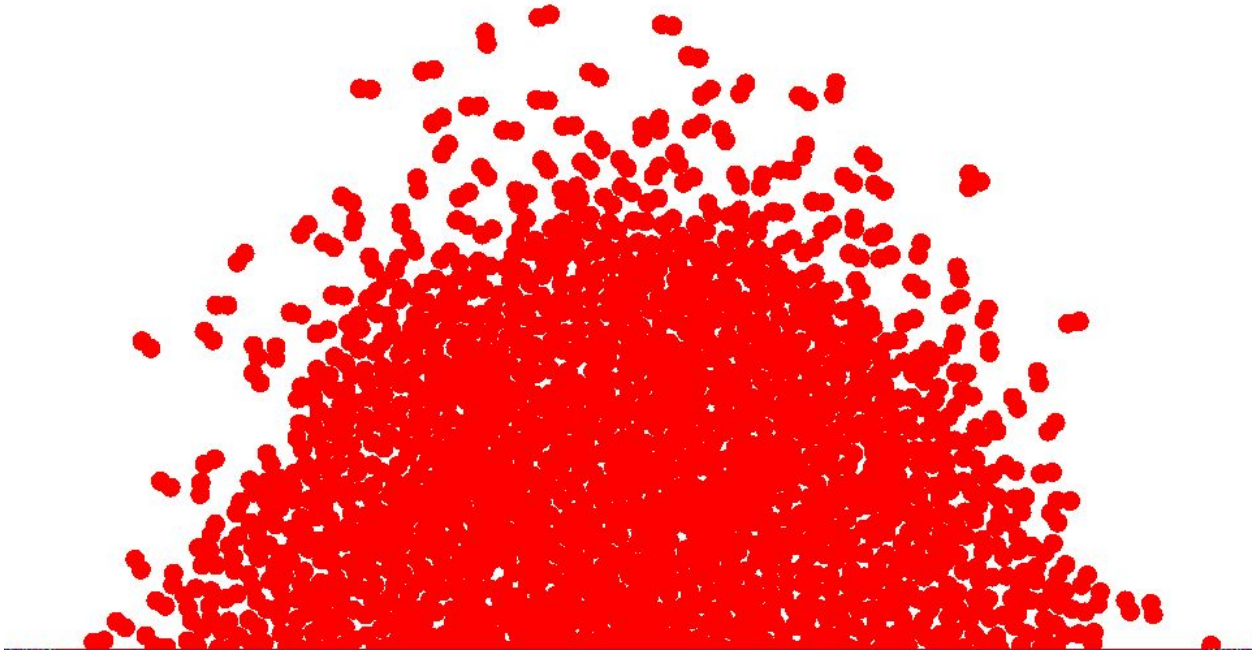


Fig 2. Intermediate configuration with particles falling due to gravity

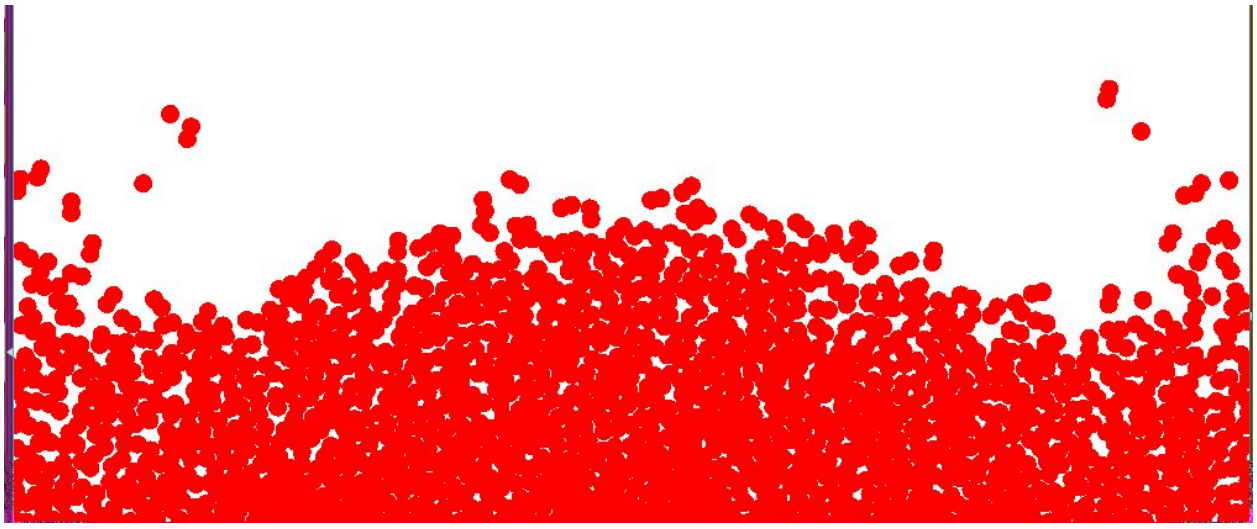
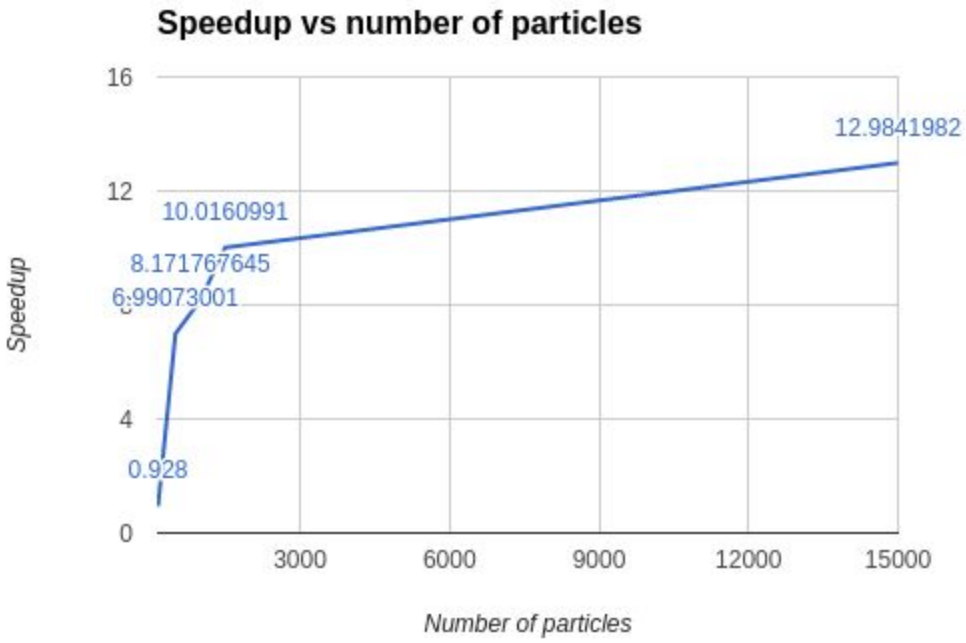


Fig 3. Final configuration of the fluid

Speedup results

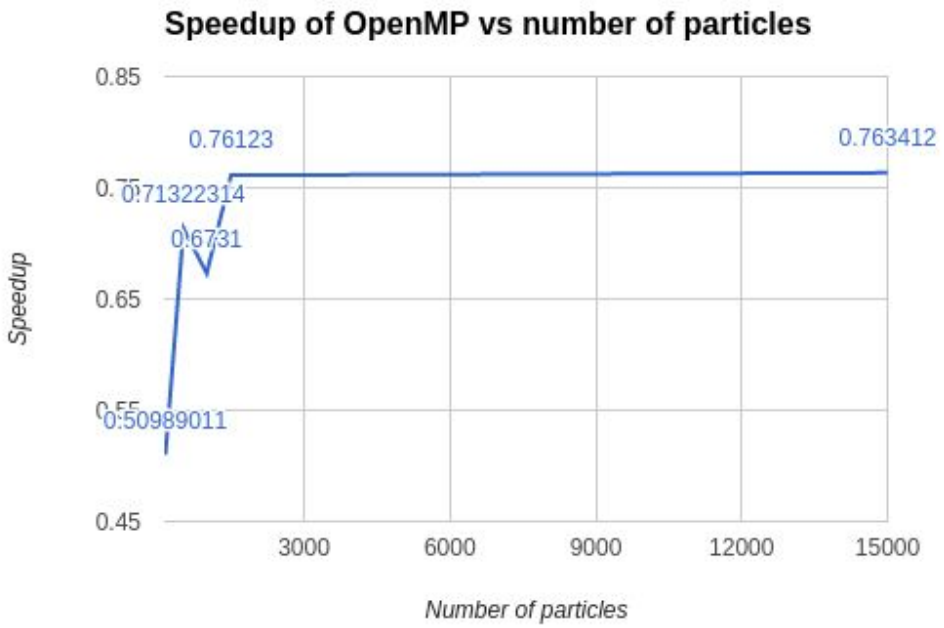
We measured the time it took to execute our critical section (i.e. the actual computation of acceleration, density, velocity and reflections) for the both serial and parallel version, and computed the speedup by using the formula given in class. This was how we measured the performance of our optimization algorithm.

Our code is structured in such a way that it is extremely easy to change fluid property values, such as the number of particles, the size of a particle, the gravitational constant, the viscosity and so on. In our CUDA implementation, it is also very easy to change the number of threads per block, since it is just a macro.



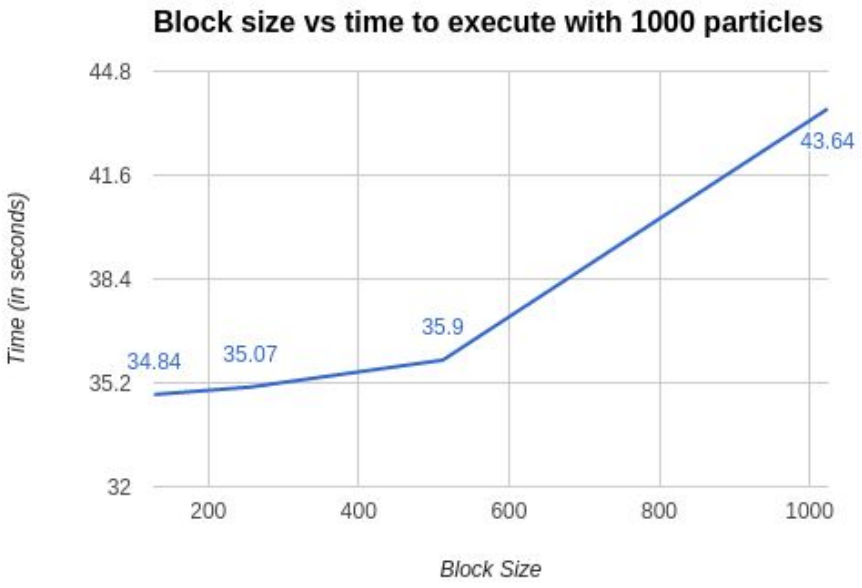
Graph 1: This graph shows the speedup that the CUDA version gets over the CPU sequential version as the number of particles increases, running on the GHC machines

From Graph 1, we see that the speedup increases significantly as the number of particles increases. Initially, our speedup is low since there is a large amount of overhead to launching threads, and due to the number of particles being small, we can easily iterate through them serially. However, as the number of particles increase, we see that the overhead of launching threads is undermined by the amount of work each thread does and hence the benefits of the speedup are seen (essentially we are noticing the span difference from O(N^2) to O(N)). We were unable to test for more than 15,000 particles since the serial version took too long to run, but we would expect the speedup to plateau at around 15x-16x.

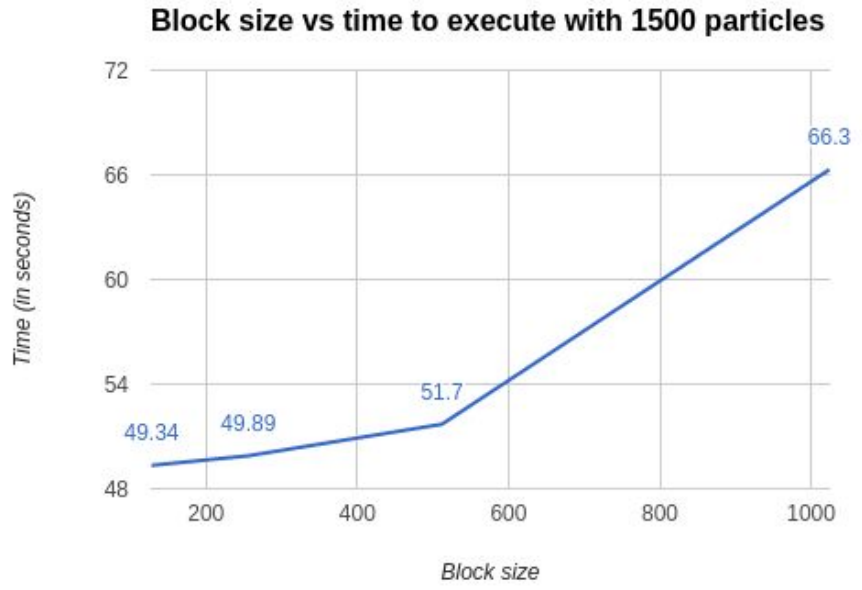


Graph 2: This graph shows the speedup that the OpenMP version gets over the sequential version as the number of particles increases running on the GHC machines

From Graph 2, we see that as the number of particles increases, the OpenMP speedup does increase. Surprisingly, we never get a speedup of 1 or higher, which means that the OpenMP version is slow than the serial version. This could be due to the high overhead associated with launching so many threads. It could also be influenced by the fact that we only have 16 execution contexts on the machine, and so we have to spend a lot of time context switching on these threads as well.



Graph 3 - Parallel CUDA version running on GHC clusters



Graph 4 - Parallel CUDA version running on GHC clusters

Graphs 2 and 3 show us that a block size of 128 is the best, since we get the smallest execution time when we use this size. In the case of the 1500 particles, this is easily explainable, since a block size of 128 has the fewest unused threads. However, with a 1000 particles, we would expect the smallest time to be from the block of size 1024, since we only need to launch one thread block. It is very surprising to see that 128 is the best block size in this case. One

explanation for this could be that with a block size of 1024, there are a lot more threads requesting memory at a time and this could cause the problem to be memory bound, which is not true in the case of 128 threads. Additionally we are not really taking advantage of any shared memory within a block, thus we don't see typical benefits of having more threads per block.

Limiting factors

One major factor that could be limiting our speedup is the fact that each of our steps is dependent on the previous one, as shown by the dependency diagram below.

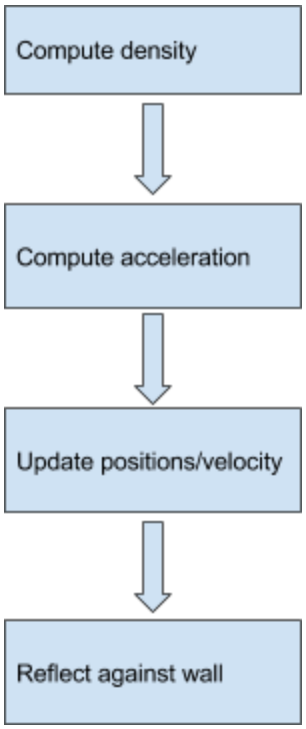


Figure 4

As a result, we can't run these steps in parallel, although we are able to parallelize each of them. Another huge obstacle is us writing to a file at every iteration, which is inherently sequential, and is thus a bottleneck. In addition, as the number of particles increases, our algorithm becomes memory bound, which once again restricts our speedup. Since we didn't have any atomic sections, any synchronization issues, and the work each of the threads were doing is the same across all threads, there should not have been any of the common issues with creating a parallel program limiting our performance.

Components of algorithm

The algorithm can be split up into the 4 components shown in figure 4. The biggest portions of the computation are in the compute acceleration and compute density sections, which constitute approximately 75-78% of the total computation in the case of fewer particles, and around 83-85% for larger number of particles. This is because there's a lot more computation involved in the first 2 sections when compared to the last 2.

Choice of Machine

We believe our choice of using NVIDIA GeForce GTX 1080 GPU's to run our code was a sound choice. Looking at the speedup graphs presented, it is clear that the most efficient version of our code was the one running CUDA on the GPU's. The reason for this primarily being that, once we had optimized our parallel algorithm, there were no dependencies or atomic sections in our code. There was just a lot of independent computation to be done on a per-particle basis. Because a GPU has significantly more execution contexts than a CPU, its architecture is inherently better to run code with large amounts of parallelism and large amounts of data.

Reference Material

We used the following website to understand the Navier Stokes algorithm
http://http.developer.nvidia.com/GPUGems/gpugems_ch38.html
We used the following website to understand the SPH algorithm and help us implement the serial algorithm:
https://www.cs.cornell.edu/~bindel/class/cs5220-f11/code/sph.pdf

Division of Work

Both teammates did equal amounts of work for this project.