15418 Final Project Report

GPU Fluid Simulation

Logan Kojiro and Oscar A Ramirez Poulat

URL

https://oadrian.github.io/GPUFluidSimulator/

Summary

We implemented a fluid simulator using Smoothed Particle Hydrodynamics sequentially, in parallel with OpenMP, and in parallel with CUDA. The purpose of this project was to explore the relative performance of different SPH implementations using diverse hardware, as well as acceleration techniques on GPU. This project was ran on two computers: one has an AMD Ryzen 7 3700x 8-Core Processor running at 3.59 GHz and an NVIDIA RTX 3080 graphics card, the other one has an AMD Phenom II X6 1045T 6-Core Processor running at 2.70GHz and an NVIDIA GTX 970.

Background

There are two main approaches to fluid simulation: grid based simulation, and particle based simulation. For our SPH fluid simulation we opted to do a particle based simulation. Though the grid based algorithms have a higher numerical accuracy, particle simulations are faster because they use neighboring particles to compute pressure rather than solving systems of linear equations and mass conservation is better and more intuitive because each particle has its own mass.

The general approach of the algorithm is to calculate the contribution to acceleration of each particle from 1) the pressure from its neighbors, 2) the force of gravity, and 3) any external forces. To calculate the forces involved with advancing the simulation, we rely on the incompressible Navier-Stokes equations for pressure and the force due to pressure. For each time step, first calculate the density at each particle with the equation:

$$\rho_i = \sum_j m_j W(r_i - r_j, h)$$

Where mj refers to the mass of the particle at position rj and W(d, h) is the smoothing kernel with core radius h that maps the distances between particle i and particle j to a scalar between 0 and 1. The Gaussian Kernel function is

$$W(d) = \frac{1}{\pi^{\frac{3}{2}} h^3} exp(\frac{r^2}{h^2})$$

Particles further than the support radius h are not counted as affecting the particle being calculated. This offers opportunity for parallelism because it means that parts of the fluid that are further away from each other are not dependent on each other and can be calculated concurrently.

After calculating the pressure at each particle, forces are calculated in a similar way as a weighted summing of contributions from neighbors this time using the gradient of the kernel function. The force from gravity is also applied at this step.

$$f_i^{pressure} = -\sum_j \frac{m_j}{\rho_j} \frac{(p_i - p_j)}{2} \nabla W(r - r_j, h)$$

Once we know the force on each particle from pressure, we can add it to the force of gravity, calculate the overall acceleration and update positions accordingly. With a huge number of particles necessary for an accurate simulation, we can parallelize over nearby groups of particles.

After the forces are calculated, we apply a first-order forward numerical integration to find the acceleration and finally positions of the particles after each time step.

$$v_{t+1} = v_t + \Delta t * \frac{F_{total}}{\rho}$$
$$x_{t+1} = x_t + \Delta t * v_{t+1}$$

Because the kernel function already only accounts for forces that are from nearby particles, there is a lot of room for parallelism in this algorithm. Both the pressure and force calculations are of order **O**(N^2) and could benefit greatly from increased parallelism. With clever sorting of the particle array, we are also able to find good spatial locality for particles that are close to each other. For each particle, though it does have to read data from all of its neighbors, it will only write to itself which means that the work for each particle can be performed in parallel within the same time step.

The main data structures for our algorithm are: a Particle array, a z-index grid and a compact z-index grid. The particle array is host to all the particles in the simulation and each particle has to keep track of the following fields:

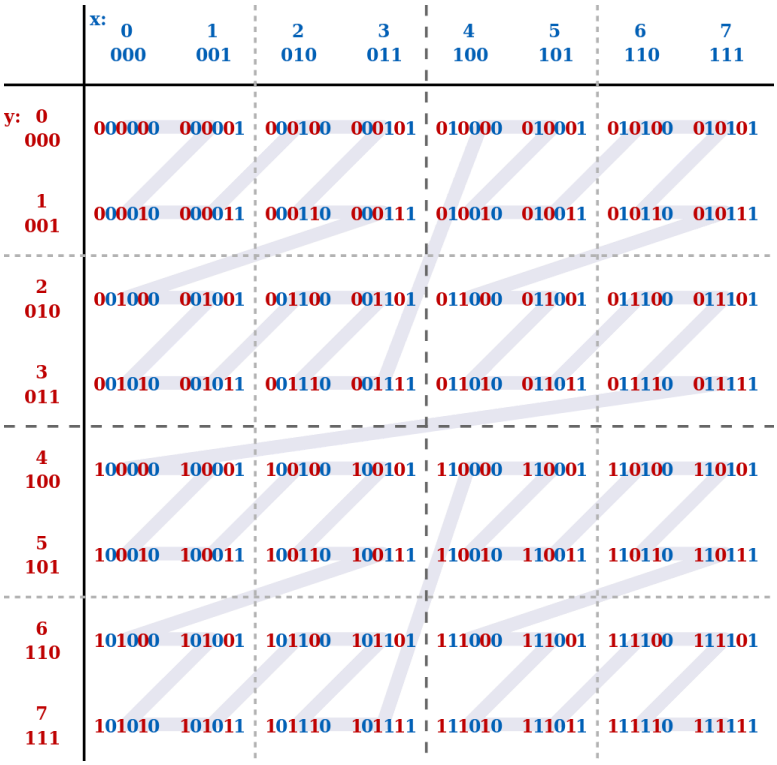
- index: this is the particle's index in the OpenGL Vertex Buffer Object array for rendering
- position: the 3d position of the particle in the bounding box
- velocity: the velocity vector of the particle
- delta_veloctiy: the change in velocity due to particle collisions
- force_press: the force of pressure the particle is under due to the neighboring particles
- force_visc: the force of viscosity due to the particle interacting with neighboring particles
- mass
- density
- pressure
- radius

- collisions_count: counts the number of particles that collide with this particle
- zindex: the index into our z-index grid

Similar to the Barnes Hut problem we saw in class, the particles in our simulator are constantly moving and thus we need a data structure that will allow fast spatial lookups. To accomplish this we opted to use a strategy similar to the one in Goswami, et al. where they use Z-indexing to create a grid where neighboring grid blocks are contiguous in memory. This allows us to have good locality among particles and their neighbors. We associate every particle with a corresponding grid cell by using its z-index. Once mapped to their z-index, we sort the particles array using the z index making all particles within a z-index block contiguous in memory. To create the grid data structure, for every block in the grid we keep track of the first particle in the sorted particles array that is in the block along with the number of particles in that block (all the particles in the block will be contiguous in memory). This grid is parameterized in terms of the size of the bounding box and the SPH support radius such that the 26 blocks surrounding any block in the z-grid are within a support radius (only particles within the support radius of a given particle will affect it). This way when looking up the neighbors of a particle we only need to iterate over 27 blocks (26 neighbor blocks + current block), as opposed to the entire grid.

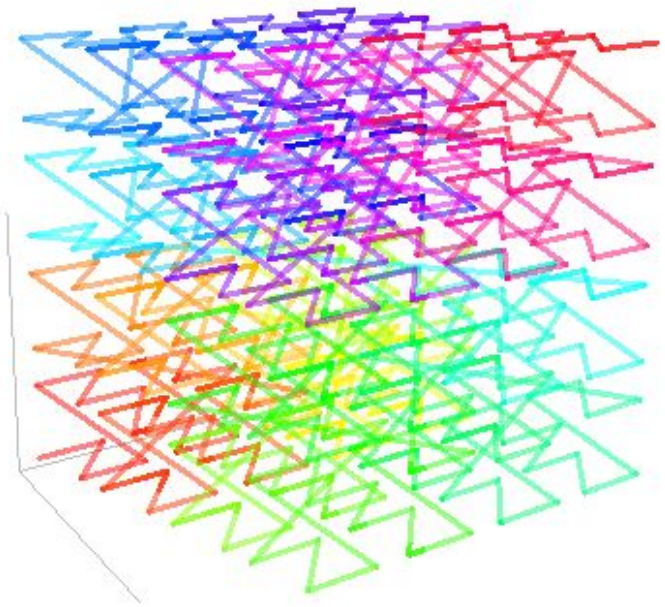
In order to maximize the utilization of the GPU we also incorporated a compact version of the z-index array that we will explain in more detail in the next section.

Figure 1: Bit-interleaved z indices on a 2d grid



y: \ x:	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

Figure 2: Visual representation of a 3d z-indexing scheme



The application we extended calls an update function before rendering the particles at the positions in the OpenGL Vertex Buffer Object array. Thus every time the update function gets called we take in the current state of the particles as represented in the particles array and perform operations using the formulas described earlier in this section on each particle. This will give us updated force and velocity information that we use to update the particle's state. The biggest bottleneck in the algorithm is the neighbor search for each particle. A naive implementation would do this in O(N^2) but using the spatial data structures described earlier and parallelism in the form of OpenMP and CUDA we can improve the runtime of this algorithm significantly. This problem is highly parallel, given that the computations on every particle are completely independent of each other. The computations are performed using Forward Euler integration i.e. the next state is only dependent on the previous state of a particle and its neighbors. The locality of the problem is created by sorting the particles by their z-index, usually placing particles that will be interacting with each other close in memory. The major dependencies are in the creation of the z-index grid arrays, given that several particles could map to the same block and we have to deal with concurrent updates to these blocks. This

created several challenges in our implementation because the number of spacial blocks in the simulation do not necessarily correspond to the number of CUDA blocks that are launched.

Approach

We implemented 3 versions of the SPH algorithm, a sequential O(N^2) algorithm, a version using OpenMP threads and another entirely using CUDA kernels. The general structure of our update consists of the following:

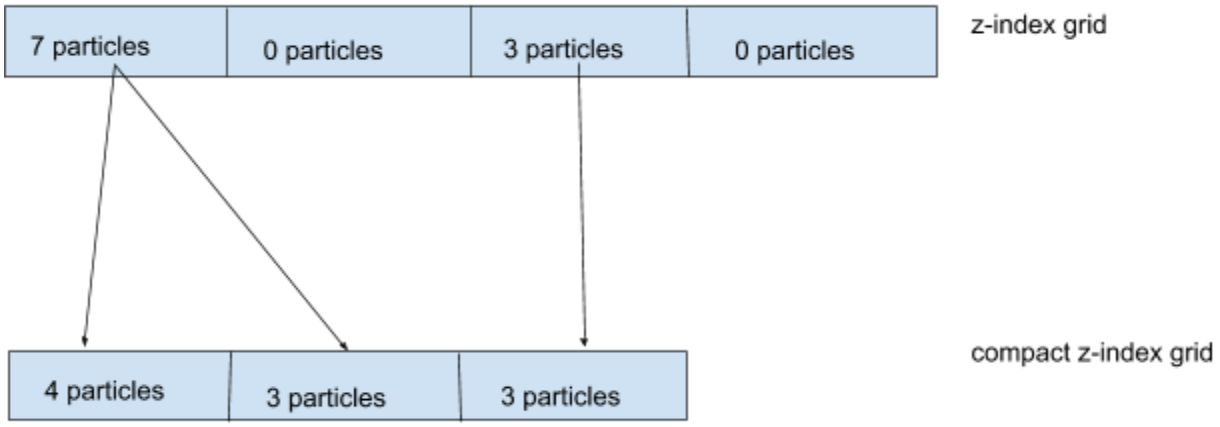
1. Map each particle to a grid block using the z-index of the particle in the grid (OpenMP, CUDA)
2. Sort the particles according to their z-index (OpenMP, CUDA)
3. Create the z-index grid array, each block in this array keeps track of the index in the sorted array of the first particle and the number of particles in the block (OpenMP and CUDA)
4. Create the compact z-index grid array. Same information as the non compact version but each block is capped at a certain number of particles for better work distribution (CUDA)
5. Compute densities (Sequential, OpenMP, and CUDA)
6. Compute forces (Sequential, OpenMP, and CUDA)
7. Compute particle collisions (Sequential, OpenMP, and CUDA)
8. Integrate system using forward euler (Sequential, OpenMP, and CUDA)
9. Update OpenGL VBO (Sequential, OpenMP, CUDA)

**** Describe how the OpenMP mapped work to threads

In order to translate this algorithm to the GPU we had to incorporate an additional data structure: a compact version of the z-index grid. Over the course of the simulation some grid cells in the z-index grid will be empty and others will be extremely full. In order to dispatch work

to the GPU more efficiently we map the original z-index grid array into CUDA blocks that are capped at a certain number of particles. Calculating this was also done in parallel with one worker thread per particle. Each particle calculates how many CUDA blocks will be created for z-indices less than its own, giving it it's index into the new array and atomically incrementing the particle count for that block. See figure 3 below for an example where the number of particles is capped at N=4 for each CUDA block.

Figure 3: mapping of z-index grid to CUDA blocks via the compact z-index grid



After this compact grid is created, one CUDA block is created for each block with threads equal to the maximum number of particles allowed in a single block. Moreover the particles in these compact blocks share the same neighbors, thus we iterate over all the neighbors of a given block and for each neighbor we calculate the interactions of the particles in the block with the neighbor.

In order to implement the neighbor sharing algorithm efficiently we use a shared memory array that is as large as the number of threads per block. We process the neighbors in batches, during each iteration each thread in the block is in charge of bringing in one neighbor into the shared memory so that all the other threads (particles) can interact with the neighbors. We use

the neighbor sharing algorithm to accelerate our density, force, and collision computations on the GPU.

We performed several iterations of optimizations for the CUDA implementation, for instance our initial implementation to create the z-index grid array used atomicMin and atomicInc to set the start and numParticles fields respectively, however we found out that this had poor performance due high contention since it could be the case that many particles try to access the same fields. This was essentially serializing our cuda kernels and thus was not as performant. It had so poor performance that our app was frozen and stopped responding. We devised a better solution that removed the need for the atomicMin function and this was much better as it no longer stopped. We had a similar issue when creating the compact z-grid where initially we used an atomicMin and atomicInc as well. In this case we managed to remove both atomic calls.

This project was an extension of the existing NVIDIA Particles Sample that comes with the installation of CUDA. The original used a basic particle system but had a nice interface to the rendering system in OpenGL, which we had no experience with. So we decided to strip out the particle system and replace it entirely with our own implementations of the SPH algorithm. The original sample can be downloaded from: CUDA Samples :: CUDA Toolkit Documentation (nvidia.com)

Results

To record the results from our project, we ran 3 separate implementations of the SPH algorithm on our two different machines. We recorded the performance of each of these on different numbers of particles and recorded the time spent in different sections of the program.

Data

Machine 1:
6-core processor @2.70 GHz, NVIDIA Geforce GTX 970 (4GB vram, 1664 CUDA cores), 8 GB RAM

Machine 2:
8-core processor @3.59 GHz, NVIDIA Geforce RTX 3080 (10GB vram, 8704 CUDA cores), 16 GB RAM

Figure 4: Execution times from machine 1

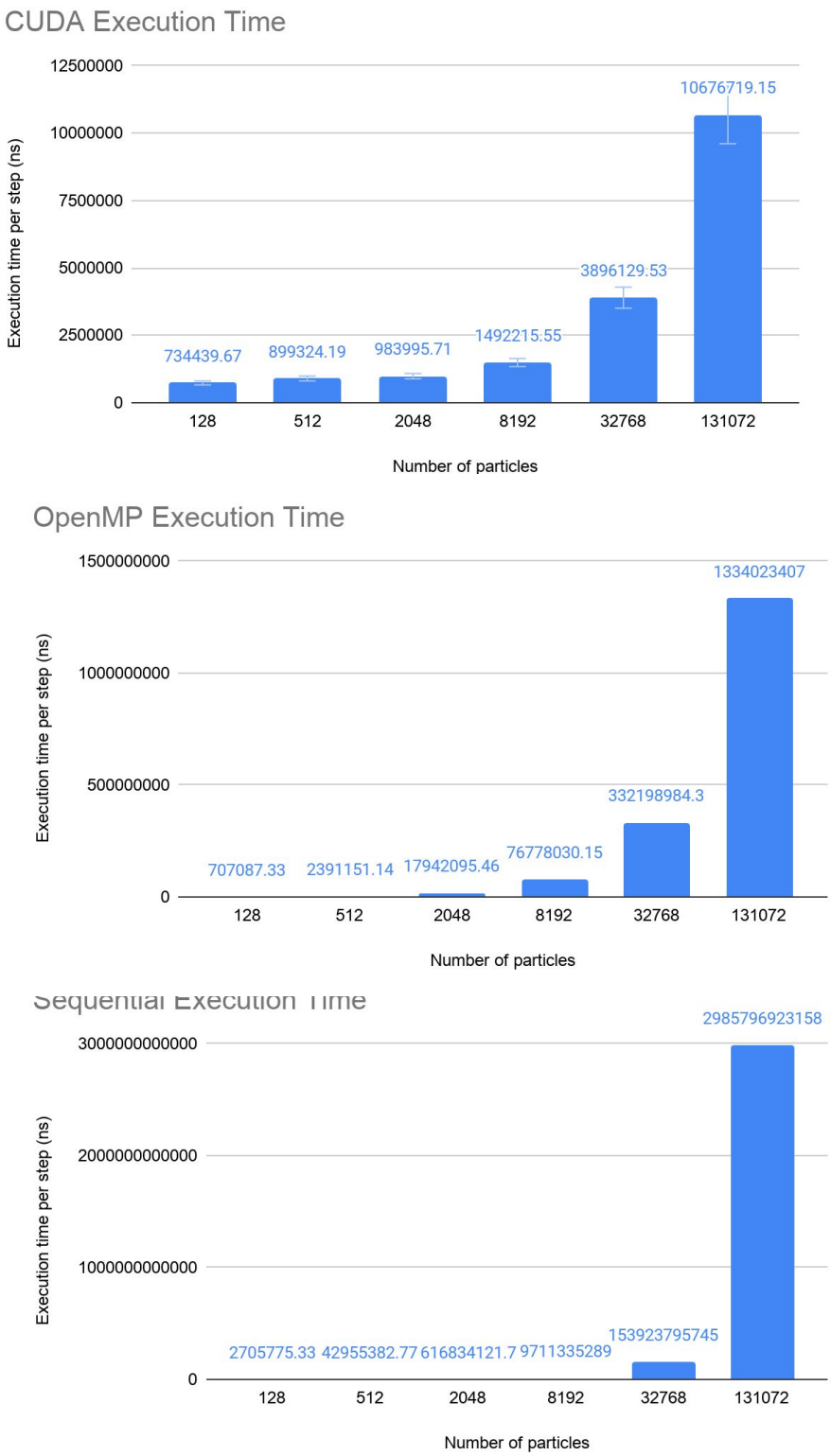
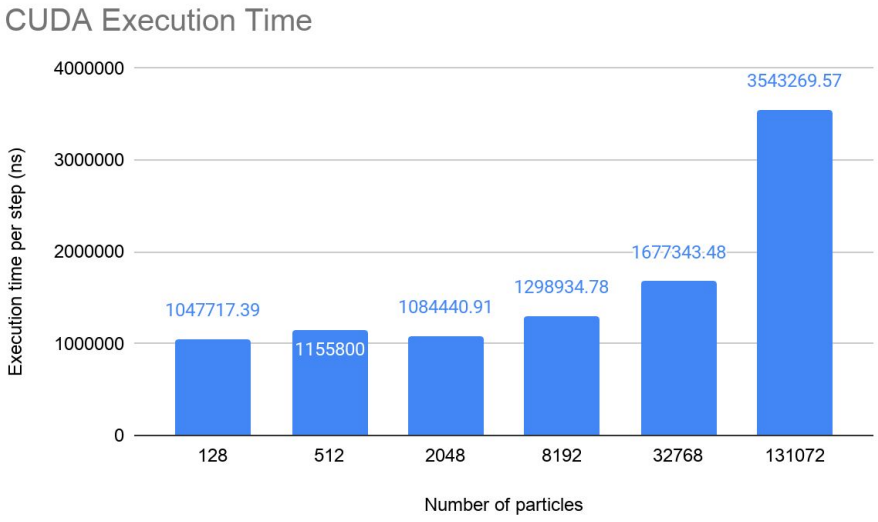


Figure 5: Execution times from machine 2



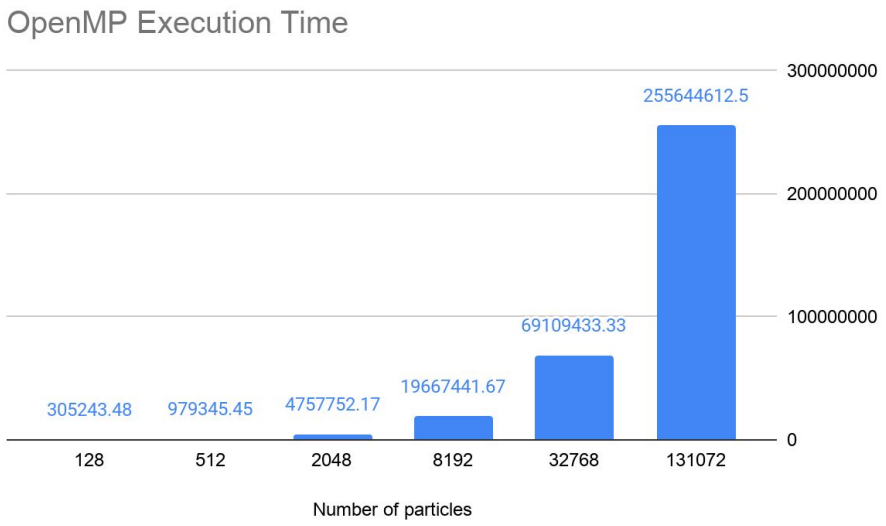
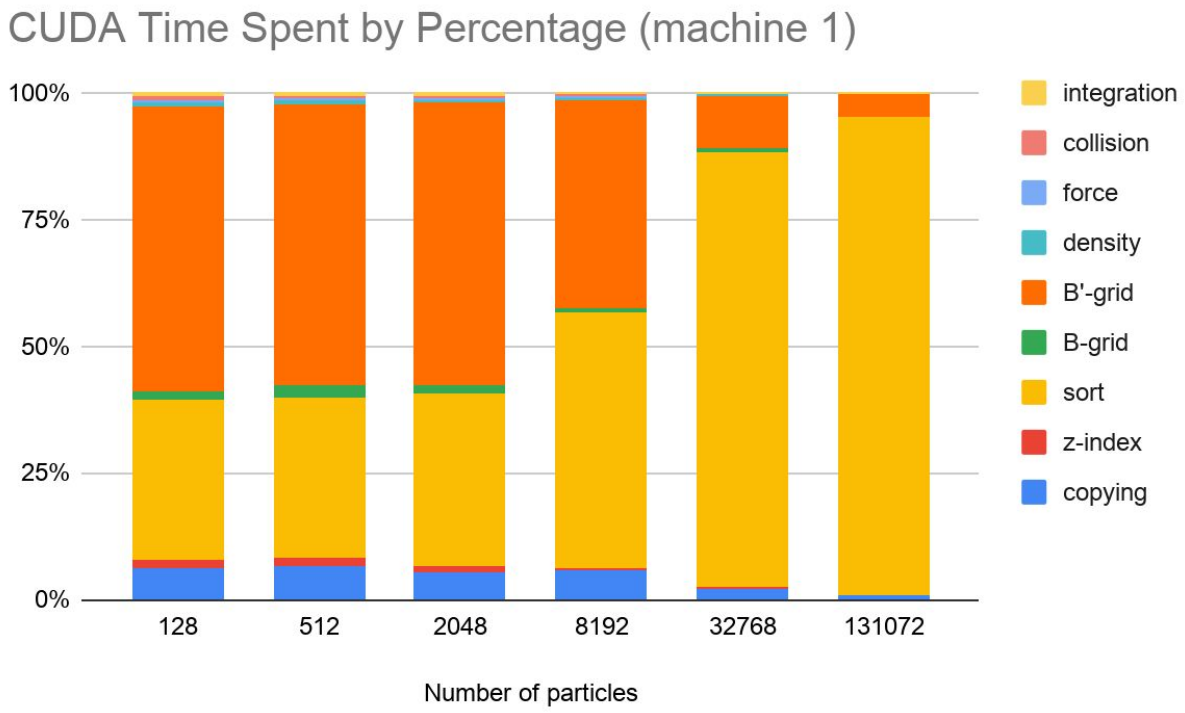


Figure 6: Execution times by percentage CUDA



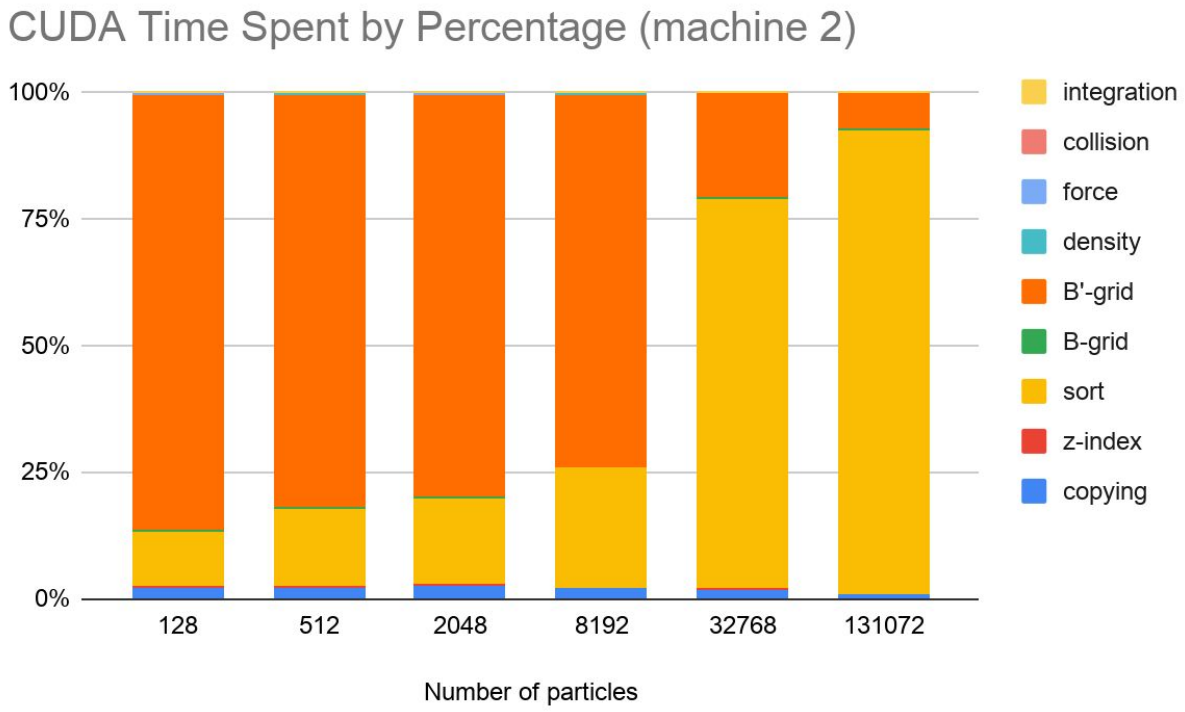
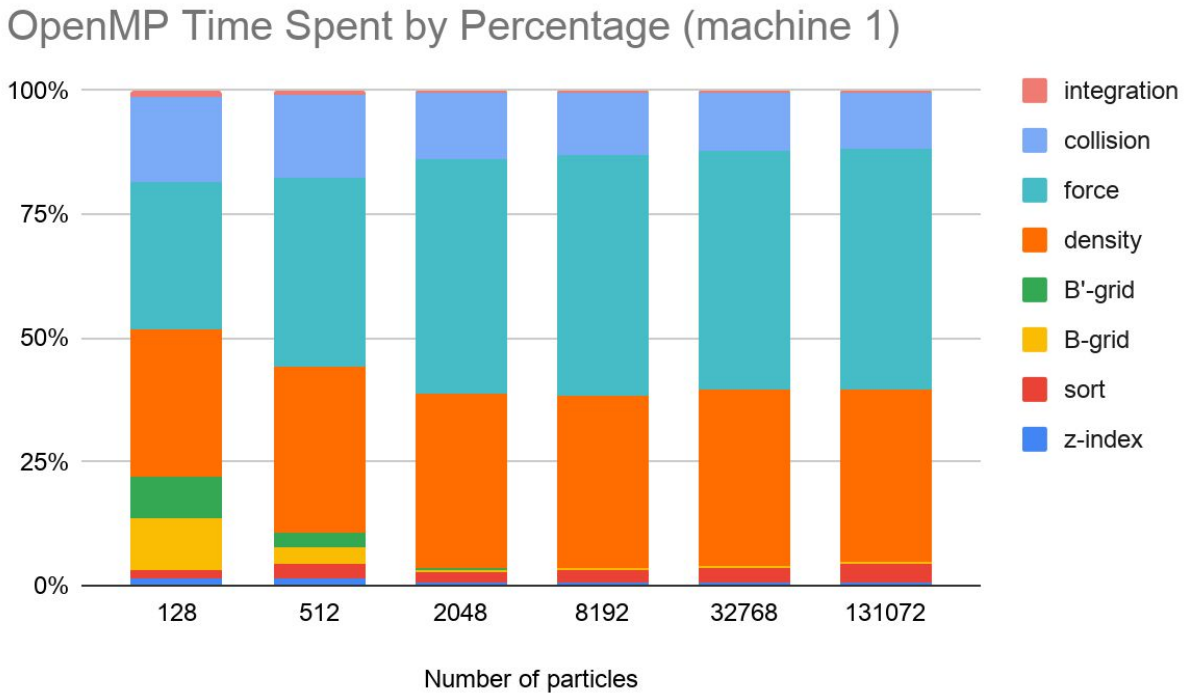


Figure 7: Execution time by percentage OpenMP



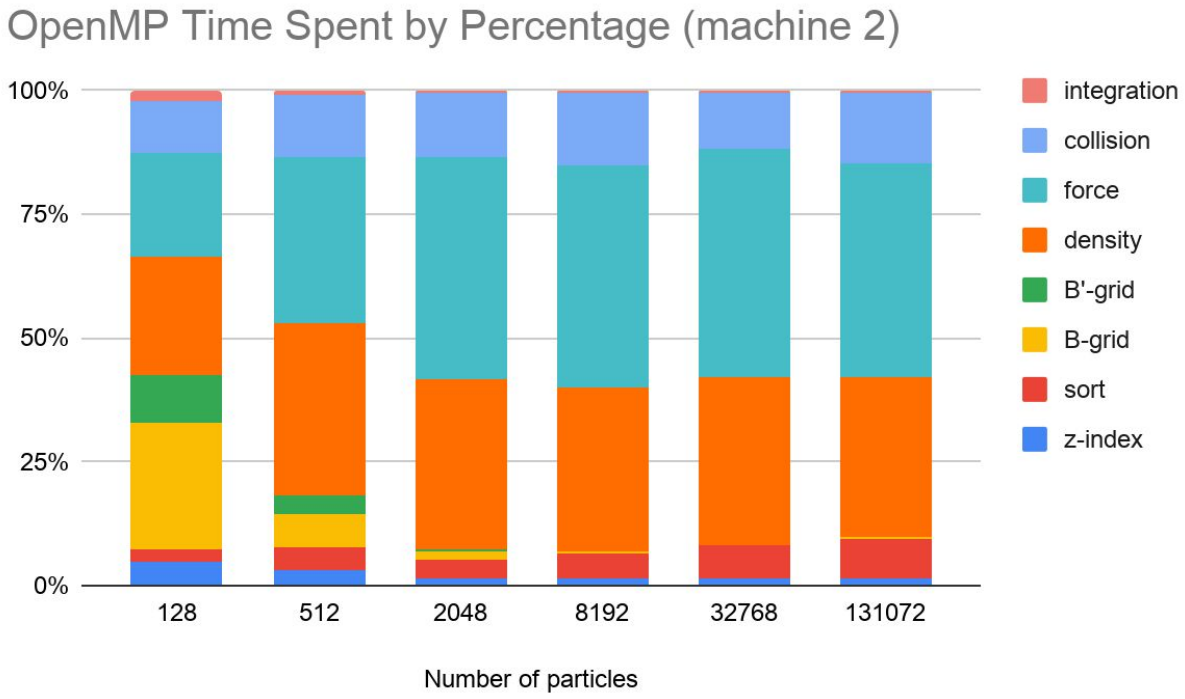
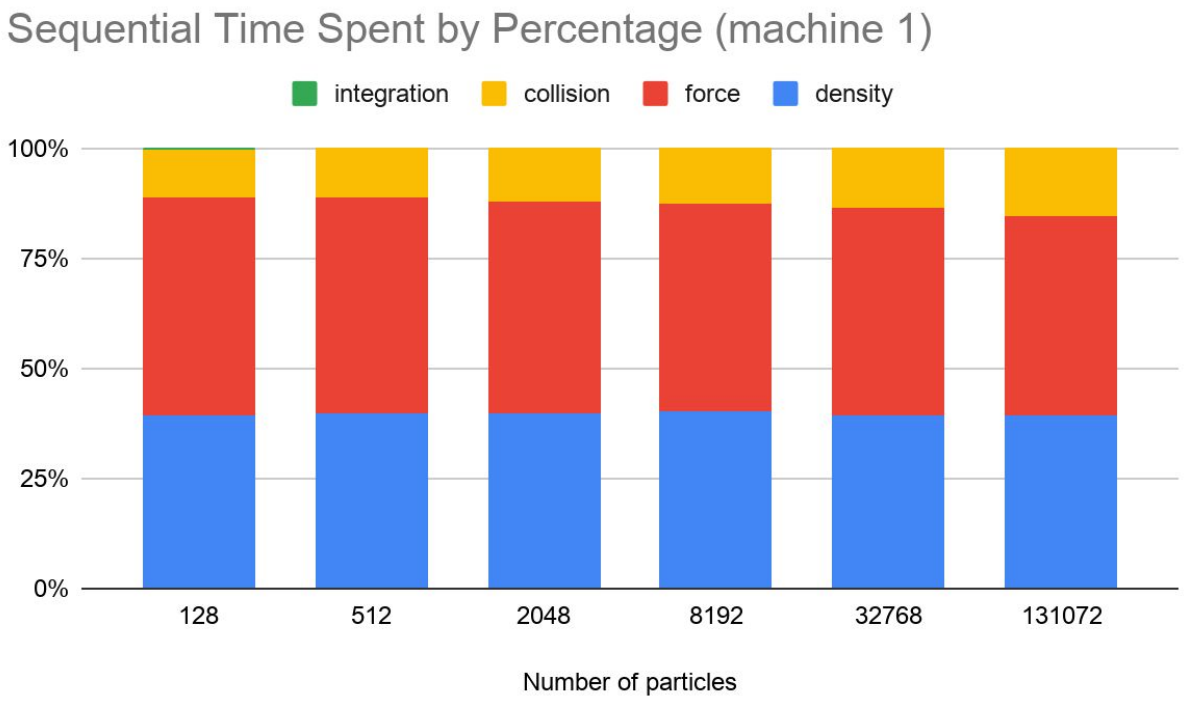


Figure 8: Execution time by percentage Sequential



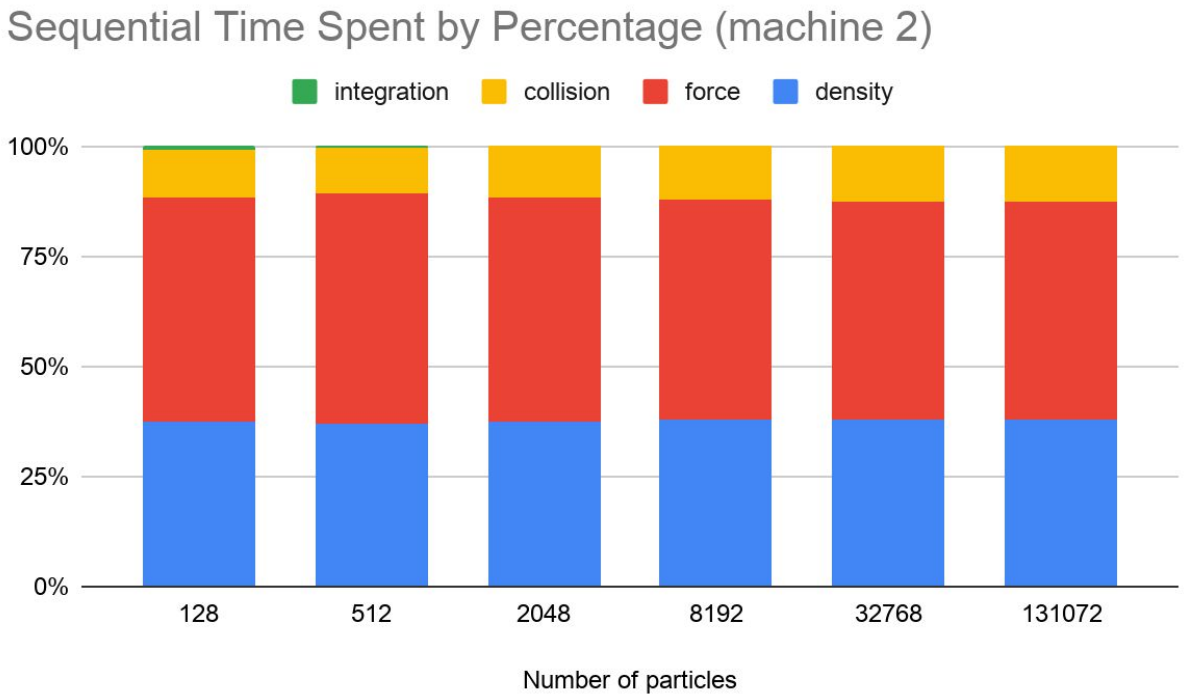
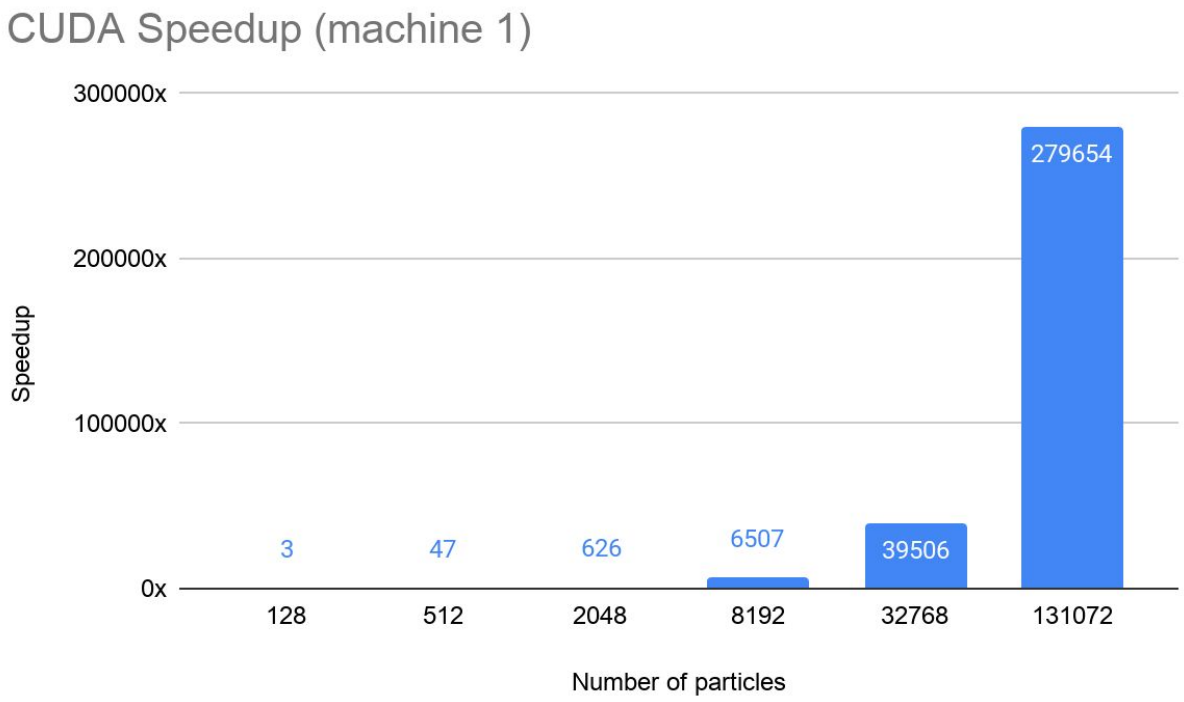


Figure 9: CUDA speedup



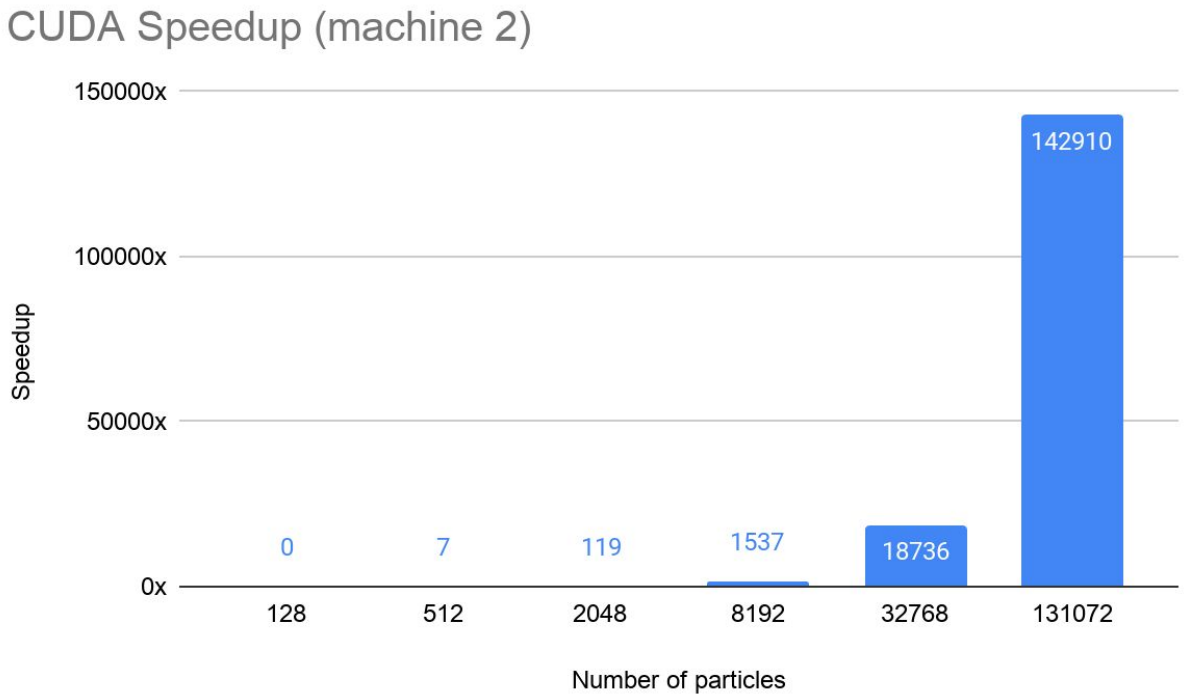
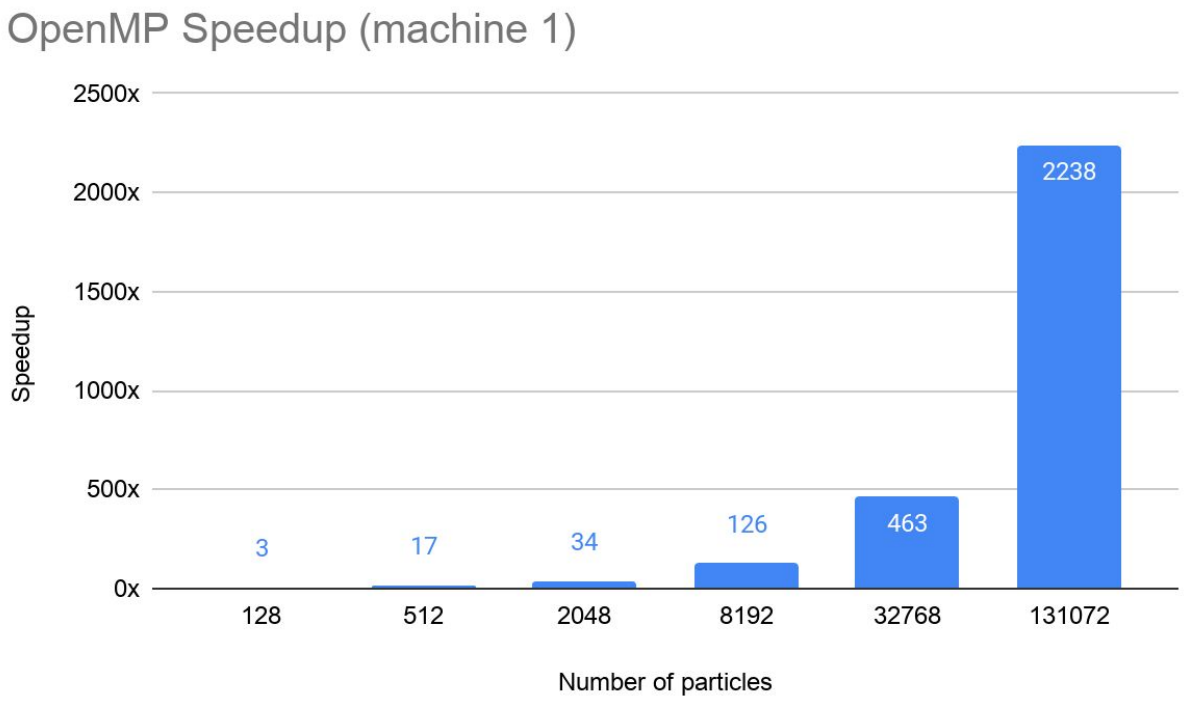


Figure 10: OpenMP speedup



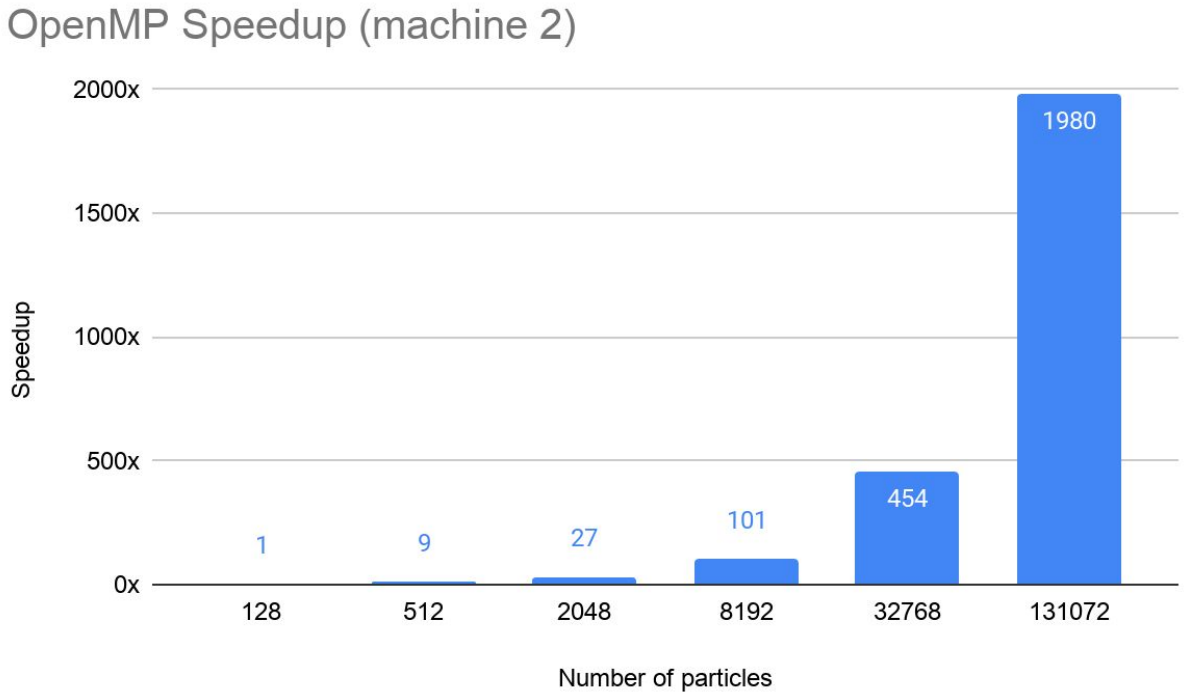
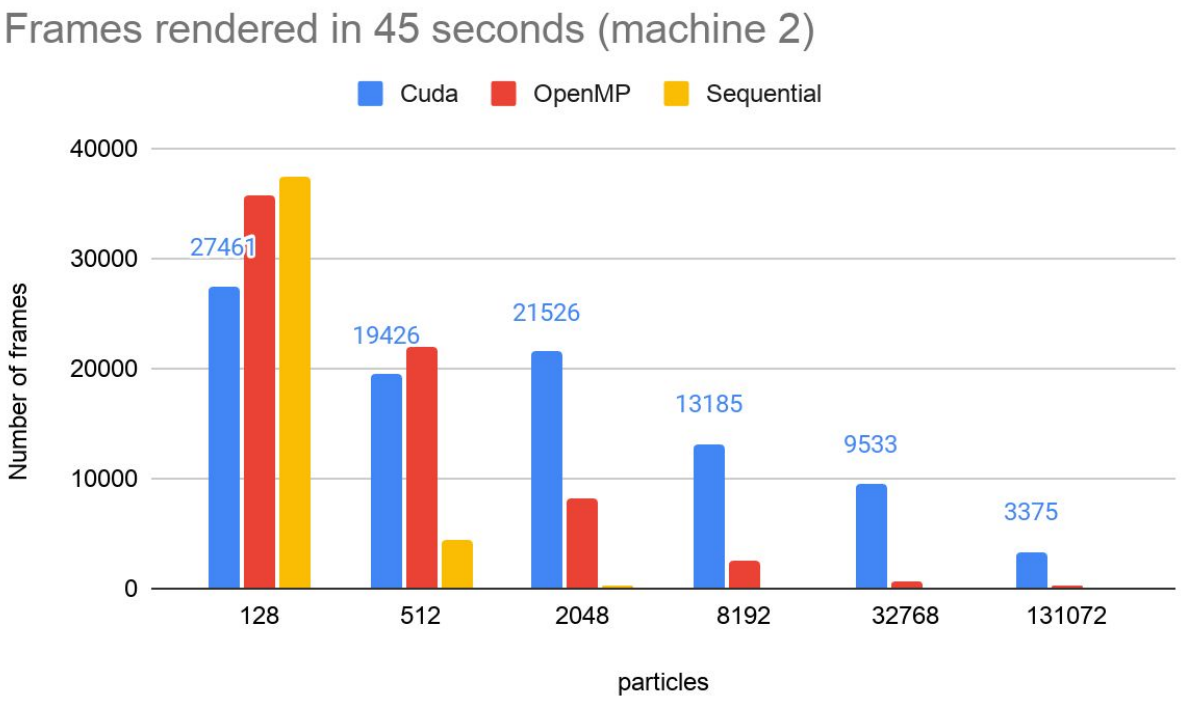
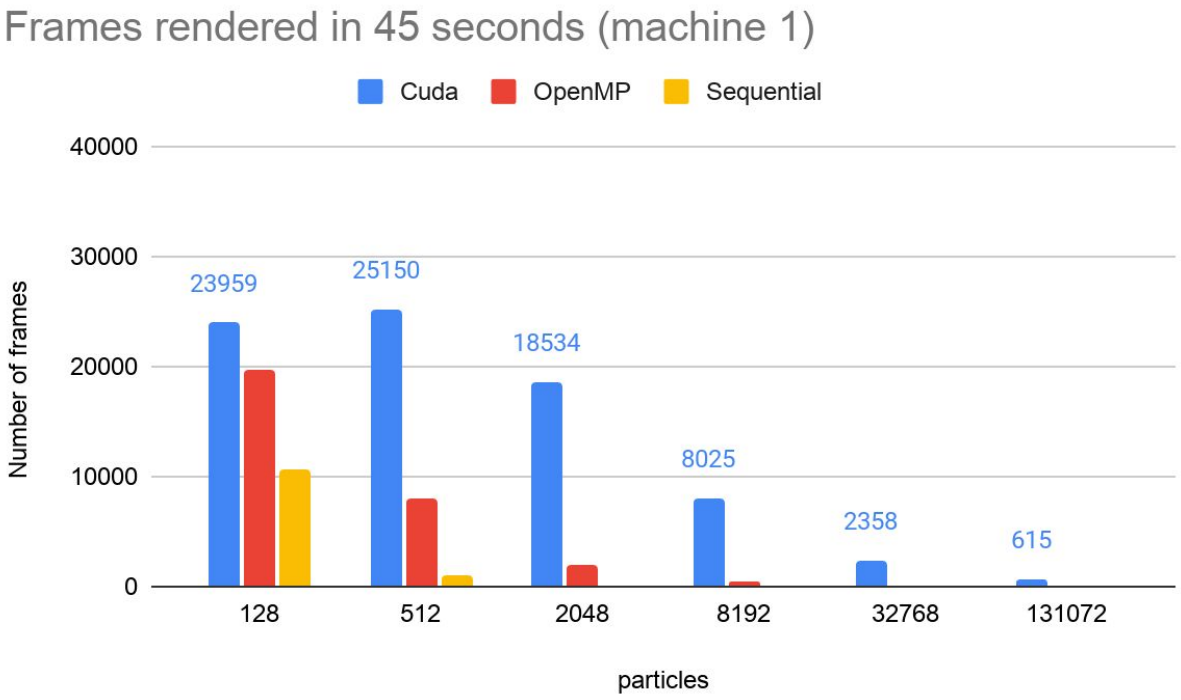


Figure 11: Time constrained Scaling





Analysis

One of the more interesting differences between the machines is surprisingly that the faster machine has a significantly worse speedup than the slower one. The reason for this is that in the slower computer the CPU is much slower relative to the GPU in that machine, which can be seen reflected in the other graphs. Notably, in figure 6 we can see that the speed of the CPU and that of the memory probably are why the copy operation takes up a much larger portion of execution time on machine 1 compared to machine 2.

One thing that is clear on both machines is that as the number of particles increases, the bottleneck in performance increasingly becomes the sort of the particles for the CUDA implementation. We used the thrust library's built in sort function for our project, and we suspect that this is one of the few places in the code where all the particles depend on all the others to get sorted. We did not look into the details of thrust's sorting algorithm, but it is possible that we could improve speedup by implementing our own parallel radix sort, which should have a linear complexity.

A big reason why sorting is not the bottleneck in the OpenMP implementation is the number of particles compared to the number of available processing units. On our CPUs there are between 6 and 16 concurrent threads running, versus the thousands available on our GPUs. Because of this, the particles have to be processed in many batches which takes much more time than std::sort. Despite this, both machines saw excellent speedup results for the CUDA code. Both are hundreds of thousands of times faster than the sequential implementation.

The speedup of our OpenMP implementation was very consistent. Looking at figure 7, we can see that as the number of particles increases, the two order N^2 operations begin to take up most of the execution time, which is to be expected.

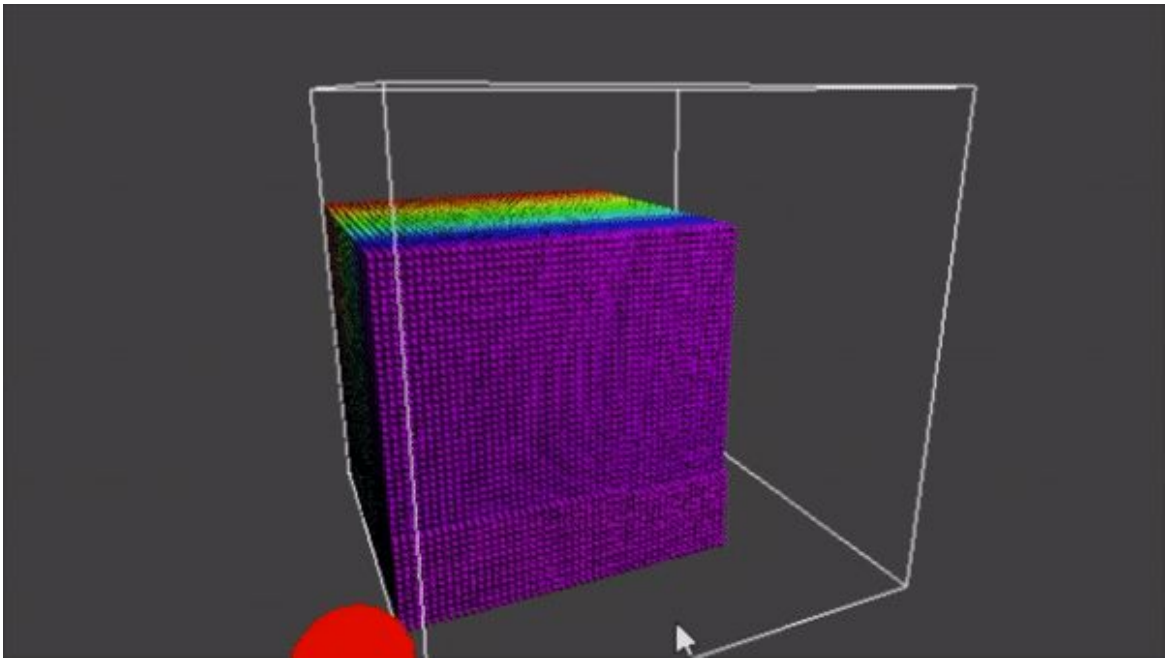
It is also worth noting the nearly constant execution times of the sequential implementation due to each of the steps being N^2 and using the same loop with minimal arithmetic.

Another important aspect of our implementations is the time constrained speedups. Especially on Figure 11 machine 2 we can see that between 128 and 131072 particles we get a near linear time constrained speed up on our CUDA implementation, whereas for OpenMP and more so on sequential the dropoff is near exponential. Another interesting aspect of the figure 11 graphs is how good the CPU is. We notice that on very small particle count such as 128 the CPU sequential implementation is noticeably faster than the CUDA and OpenMP, this goes to show that the overhead of managing many threads, in the case of OpenMP, and the overhead of switching to CUDA and the fact that there is little parallelism negatively affects these implementations. Even at 512 where the dropoff of the sequential implementation is vast, the OpenMP version outperforms the CUDA version. It's not until we reach the very high particle counts where we can see how well the CUDA implementation scales. The story on machine 1 is quite different, in this case the CPU is not as fast and thus the CUDA implementation reigns supreme in every particle count. The scaling is less linear than machine 2, but it still scales well when time constrained. Similarly the OpenMP and sequential versions have a steep exponential drop which is expected.

Overall, we were very pleased with the results. We achieved a very good speedup on our CUDA implementation that allowed us to run very large simulations

(100k+ particles). It was very satisfying to watch the number of particles we could run in real-time grow as the project progressed.

Figure 11: sped up final simulation result (100k particles)



References

Goswami, Prashant, et. al. " Interactive SPH Simulation and Rendering on the GPU". Eurographics/ACM SIGGRAPH Symposium on Computer Animation. 2010.
link: https://www.ifi.uzh.ch/vmml/publications/interactive-sph/InteractiveSPH.pdf

Colin Braley, Adrian Sandu. "Fluid Simulation For Computer Graphics: A Tutorial in Grid Based and Particle Methods". Virginia Tech.
link: https://cg.informatik.uni-freiburg.de/intern/seminar/gridFluids_fluid-EulerParticle.pdf

We used CUDA and OpenGL examples from:
https://docs.nvidia.com/cuda/cuda-samples/index.html#graphics