CS 470 Final Project: Particle Simulation with CUDA

Lauren Hartley, Jack Ball, Ye Hun (Samuel) Joo, and Josh Kuesters

May 4th, 2023

https://github.com/ball2jh/particle-simulator/

1 Introduction

The Parallel and Distributed Systems (CS 470) final project was a semester-long project where we were able to use CUDA and GPU programming to parallelize and significantly increase the performance of a serial program. For our project, we created a simple particle collision simulation with OpenGL. Using CUDA to parallelize collision simulations is not a novel concept, but we proposed creating our own simulation so we are able to understand the entire process of creating a serial program and parallelizing it.

Our goal was to parallelize our serial simulation with CUDA so it could run on increasingly large particle sizes with a reasonable frame rate. Collision simulations are computationally intensive, and serial implementations experience a drop in frame rate when rendering large amounts of items. By parallelizing these computations, we aimed to speed up rendering times to allow for more objects to be rendered while maintaining a consistent framerate.

We fell short in our final project check up due to having some difficulty trying to understand OpenGL and CUDA, however we believe that we have made some significant effort in our program.

2 Current Program

Our program, as it currently stands, can display a window with a user specified number and size of the particles that deflect off of the border and surfaces of any other particles. When in process, our collision detection and replacement functions allow for the particles to have dynamic movement without errors after collision.

The program utilizes a function that calculates the distance from each particle's center to the radius and will interpret collisions if any other tangible element enters that zone. Upon collision, the program will perform a check for any intersections between their borders and repositions each particle into the nearest non-overlapping positions. Additionally, it will perform an update to the velocity of each particle altering their vector to one that is inverted at the point of collision. Similar error checking is performed at the point of collision with the border.

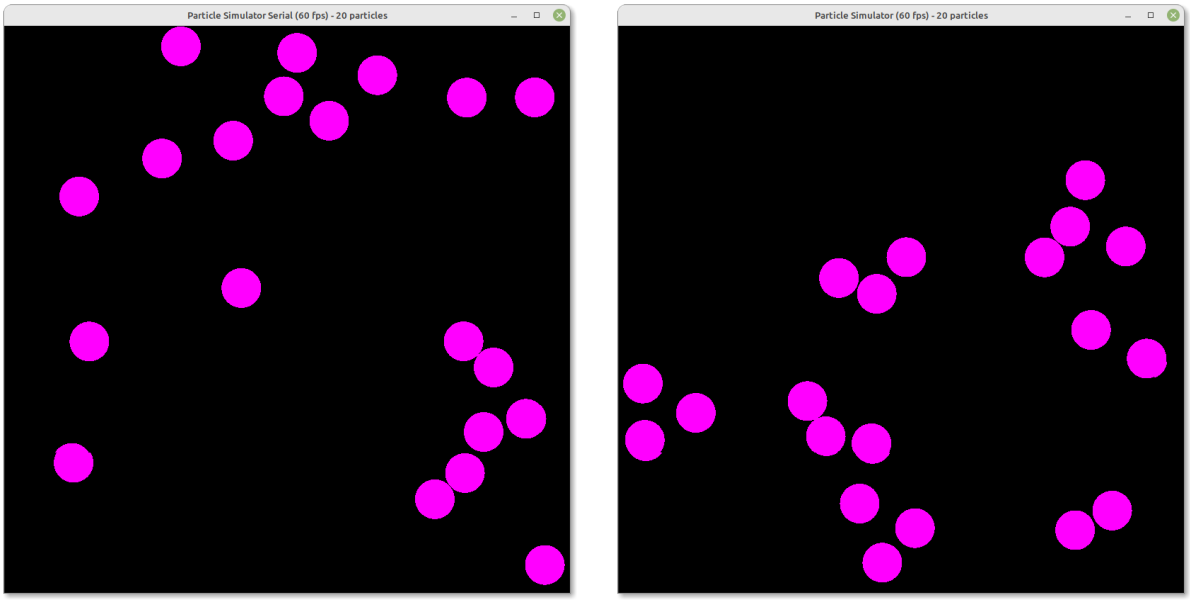

Figure 2.1 - Serial and CUDA programs side by side with 20 large particles.

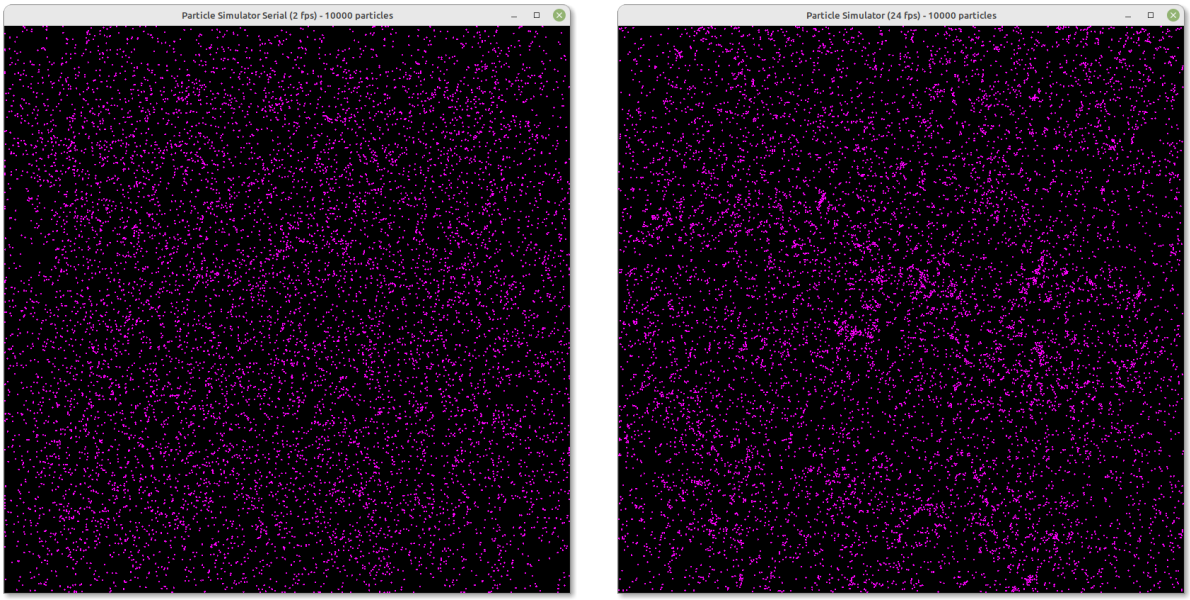

Figure 2.2 - Serial and CUDA programs side by side with 10,000 small particles.

The simulation uses a relatively basic collision resolution. Initially, particles will be spawned with a random velocity, position, mass, and direction. The particles will ricochet off the walls and other particles before slowly losing momentum.

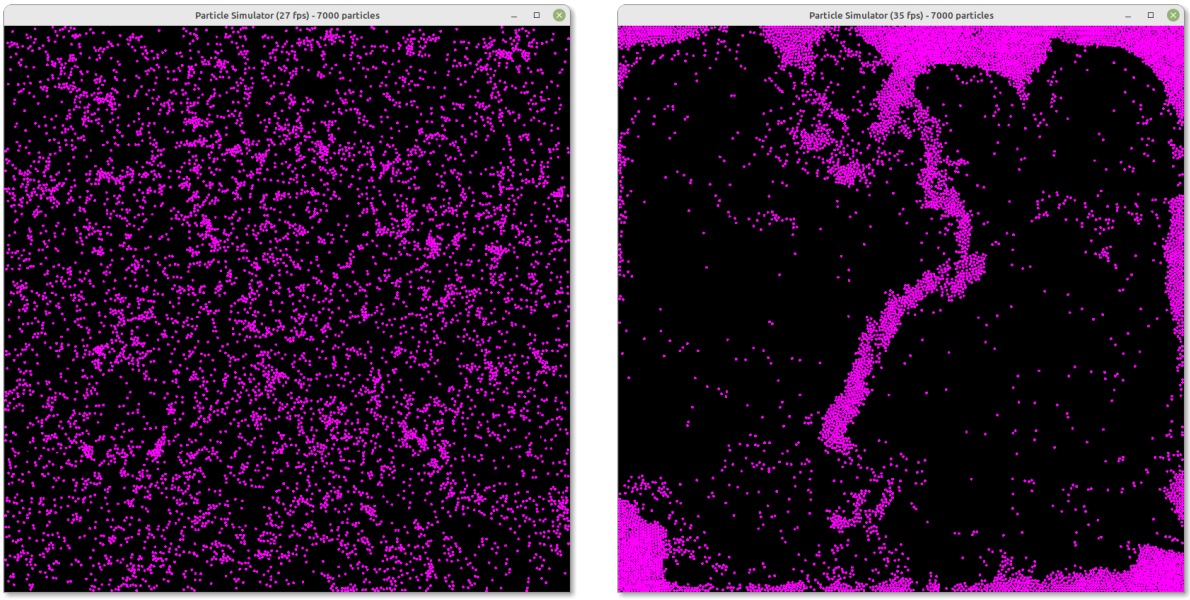

Figure 2.3 - Coalescence of 7,000 particles after approximately two minutes.

3 CUDA Kernels

In our program we implemented two kernels in order to run the particle collisions more efficiently than the serial version and they have produced some significant results as we discussed in figures 3.1 and 3.2 above.

3.1 `checkCollision`

The first CUDA kernel is implemented on the `checkCollision` method that checks whether two articles are about to collide with each other and resolves their collision. The kernel works by first getting the block index along the x-axis of the grid of blocks that the kernel is running. Next it gets the thread index along the x-axis in the block. Then it gets the global index for each thread using the formula of block index multiplied by the block dim (represents the dimension of threads in a GPU kernel) and adds the thread index. Then using this global index it uses a for loop that checks if i and j (j = i + 1) collide using the collidesWith method and then using the resolveCollision method if they are gonna collide.

After running our program with the profiling tool NVIDIA insight, we found that the majority of the speedup likely comes from the kernel. For detecting collisions, our simulation uses a brute force approach to determine whether particles are colliding. Therefore, it is expected that this kernel would be doing exponentially more operations than our other kernel.

3.2 `updatePosition`

The second CUDA kernel is implemented in the updatePosition method which, just as the name suggests, updates the position of each particle accordingly. First it gets the global index the same way as the checkCollision method by using the block index, thread index, and block dim. Next it goes into a for loop to go through all the particles and each thread generates their own random int to put into a global array. Then it generates a random float value that is scaled between 2 and 4 so that the velocity won't be too drastic. Finally, it updates the position of the particle and then checks to make sure that it isn't out of bounds by checking if it will bounce off the walls. This uses CUDA by allowing different threads to update a multitude of different particles without any major cost to the performance of the program.

3.3 `display`

The `display` function serves as the main function for rendering the particles and the host function that calls the CUDA kernels. The circles are initially rendered serially with OpenGL with a simple loop. Before the function invokes the CUDA kernels, `cudaMemcpy` is used to copy the particles in an array on the host to the CUDA device. The two kernels are then called to update the particles and check for collisions before `display` retrieves the particle data from the device. The CUDA device is then synchronized before the frame rate is calculated and displayed on the simulations window.

4 Performance Analysis

4.1 Frame Rate

With small numbers of particles, the serial program and CUDA program run almost identically. We also observed that particle size has minimal impact on performance. While comparing the frame rate on large inputs, we were able to observe a significant difference between the performance of the simulations. All of our test data was gathered through testing on the lab machines, but with better GPUs the simulation may be capable of rendering more particles.

With a relatively large number of particles with size 0.004, we observed that the serial version experiences a considerable decrease in frame rate between 1,000 and 2,000 entities. The CUDA simulation's frame rate gradually decreases, but it is able to maintain a faster frame rate for much longer than the serial version. At 7,000 particles, the serial version maintains 1 fps until around 25,000 when it is no longer able to render the vertices at all. Although the frame rate is low, the CUDA simulation is still able to display and calculate the collision of 60,000 particles with 2 fps.

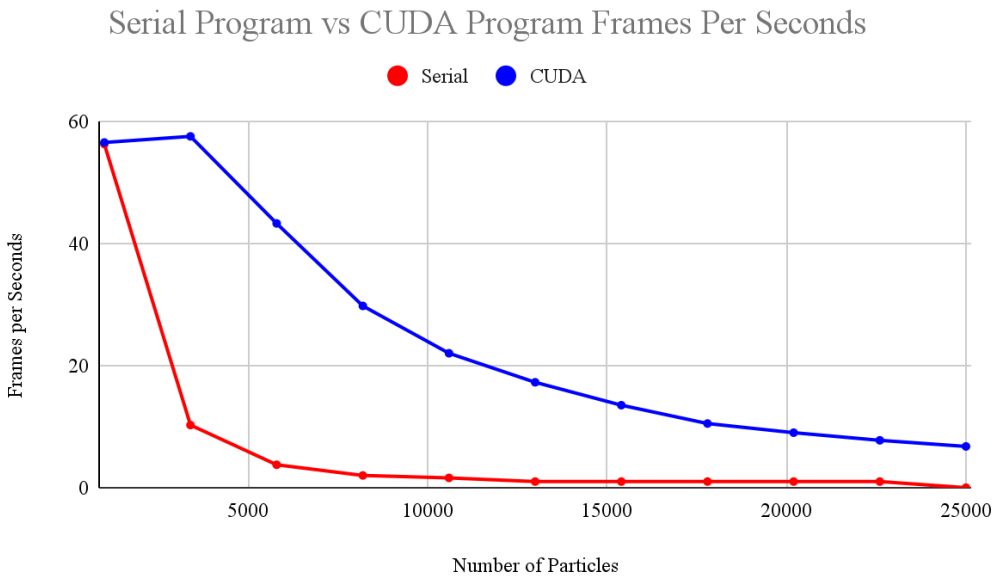

Figure 3.1 - Serial vs. CUDA FPS Plot.

Number of Particles	Serial	CUDA
1000	56.25	56.5
3400	10.25	57.5
5800	3.75	43.25
8200	2	29.75
10600	1.6	22
13000	1	17.25
15400	1	13.5
17800	1	10.5
20200	1	9
22600	1	7.75
25000	0	6.75

Figure 3.2 - Serial vs. CUDA Average FPS Data.

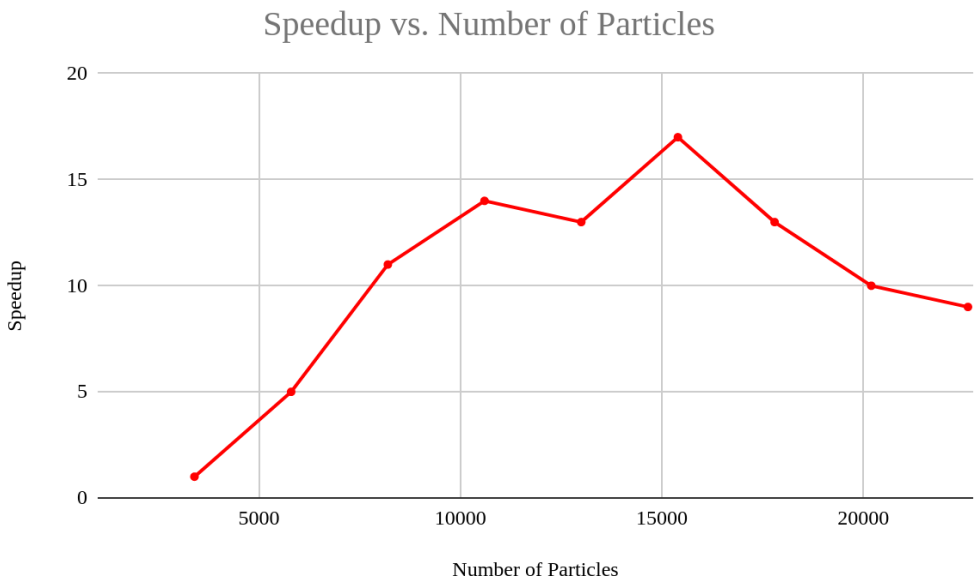

Figure 3.3 - Speedup (parallel time over serial time) for number of particles.

Number of Particles	Speedup
1000	1
3400	5
5800	11
8200	14
10600	13
13000	17
15400	13
17800	10
20200	9
22600	7

Figure 3.4 - Table of data for speedup (parallel time over serial time) for number of particles.

5 Obstacles

Throughout the course of our project, we faced many roadblocks. Combining OpenGL with CUDA has been a challenge, especially since they are both new technologies for us. For instance, we found that rendering the particles onto the screen must be done serially. We attempted to use OpenGL vertex buffers, which allow for a more efficient transfer of

vertex data between the CPU and GPU, but were unable to implement them due to time constraints. CUDA and OpenGL have libraries that handle interoperability, so this is potentially something we would be able to address with more time.

We also attempted to implement a quadtree for our collision detection, but decided to implement brute force collision detection so we could focus our attention on other aspects of the project. A quadtree would've allowed for more efficiency and likely would've improved our programs frame rate.

6 Future Improvements

Our current program is not all that we had hoped to create for this final project and there are a couple things that we believe could be added to further improve the use of CUDA.

6.1 More Entities to Collide With

One idea is to add some collision for the mouse as the user moves it across the window that would make it so that if the particles collide with the mouse they will bounce off of it. We could also implement a drag element to the mouse so that you could grab a particle and move it around knocking it into other particles to create more instances of parallelism to increase the efficiency of a physics simulation. Another idea we had that is similar to the cursor idea just discussed is adding some terrain for the particles to interact with. It would be interesting to see some sort of physical presence other than the boundary of the window and other particles play some effect in the movement of the particles.

6.2 Different Environments and Particle Properties

Another idea for the particles is to have a different environment for them to float around in, for instance, if we added a form of gravity to the equation, the particles would move in an entirely different way and using the cursor physics idea discussed earlier could make it more interesting. We could also try to make the particles themselves other shapes or maybe even fluids to test out the performance that CUDA could have on the particles in larger quantities since they wouldn't take up the whole window.

6.3 Detecting Collisions with a Quadtree

As previously mentioned, we would also like to switch our collision detection from brute force to a quadtree. This would have improved our frame rate and may have allowed us to use CUDA in more interesting ways to improve our collision detection. When designing our project idea, we wanted to render much more particles than what our simulation is currently capable of, so this would be one of our primary future goals.

7 Conclusion

There are a lot of more ideas that are already out there that we could take a look into to try and implement or even try to improve using CUDA as a guide. Despite the obstacles we experienced throughout the course of our project, we were able to experience building a simple collision simulation and observe the effect of parallelism.