600086 Lab Book

Week 9 – Lab I

Date: 18th May 2023

Q1. GPU Design

The CUDA implementation works by performing the projectile motion calculations inside of the kernel function. There are 3 particle systems, consisting of 300,000 particles. Each of these allocates the correct amount of space using float3 values, and then is run on the kernel using a block size of 256 which is then divided into the number of particles per particle system to calculate the right number of blocks per particle system.

Within the kernel function, using the appropriate equations and considerations like gravity and drag on the particle, its position is altered within a while loop until the Y position of the particle is equal to or less than 0. This is because we can safely assume the particle has rather hit the paper, or hit the table when its Y meets this condition.

The kernel function also holds the functionality of calculating the angle of trajectory for each particle. This is done by first calculating the directional vector from the particle's location, to the centre of the paper ignoring the difference in Y. We then generate as float3, where each X, Y, and Z value is randomly generated between the bounds of a value in degrees. This float3 is then used to generate a rotational matrix which is applied to the directional vector to generate a trajectory for the particle which resembles that of a cone within a batch of particles.

Finally, once all the particles have hit a Y level equal to or less than 0, the program will cycle through each particle system. If the particle's position is found to be within the bounds of the paper dimensions, then the X and the Z values are rounded to the nearest whole number. And placed into the index values of the Paper Arrays for X and Y. To which then using the blend equation the paper pixel's colour value is altered based on the colour of that pixel.

Finally the colour values for each pixel is converted to RGB values and outputted as an image in the directory. To which all memory is then allocated and timings are recorded for each stage of the simulation.

Q2. CPU Design

The Rust implementation follows a similar format to the CUDA implementation, in terms of projectile motion handling.

Firstly we intiliaise the Paper object and all of its elements. And then we do the same for the 3 particle systems. After that we set up a scoped thread pool with a thread for each particle system, which then calls the run simulation function. In this function we create another pool of threads that calls the move particle function within the particle implementation for each particle. This is done using 4 threads split across all particles within the particle system evenly by NUM\_OF\_PARTICLES / 4.

Within this function we perform the same actions as discussed in the CUDA implementation. We take advantage of a directional vector, calculating this using the centre of the paper and the position of the particle and normalising the result of minusing one from the other. Which we then use alongside with the spray cone angle value to generate a rotational matrix. We can use this and apply it to our velocity vector to generate a trajectory value for our particle.

Finally within this function, we continuously iterate our particle position using the velocity value until the particle's Y value is equal to or less than 0. We know that if the particle gets to that point it has rather collided with the paper or collided with the desk.

Once all of the particles have reached a Y value of 0 or lower, we then move onto checking if the particles collided with the paper object or not. To do this we cycle through all of the particle systems, and for each particle we check if the particle's current position is within the bounds of the paper's dimensions. If it is, we then go ahead and use the blend equation to correctly apply the colour of that particle to that point on the paper. To calculate the correct pixel within the paper array. We simply round the X and Z value of the particle's position to a whole number, and then use that as the index values for the X and Y value of pixel elements within the object. This helps us to find the closest pixel to the particle, to which we can apply the colour to.

Finally we cycle through all of the paper elements, and multiply the colour values by 255 to get the RGB equivalent. Which is used to generate an image as an output to display the path all particles took during the simulation.

Q3. Performance Metrics

CUDA Performance Metrics



Rust Performance Metrics



CUDA Total Time = 1.2203059 seconds

RUST Total Time = 10.209 seconds

As you can see just from the image above, the CUDA implementation is significantly faster than the Rust implementation. I made the CUDA version based off the Rust implementation, so all of the equations and calculations are performed with the exact same methods making it a more accurate representation of speed between the two implementations.

From the results it is pretty clear that the CUDA version takes advantage of the better threads on the GPU, as the CPU threads are run through cores which have to be very generally performant. This is effective to handle all kinds of different tasks that the CPU would be tasked with. Whereas the GPU focuses its performance on more specific areas. And as such these large amounts of maths calculations in parallel are more efficiently performed on the GPU as a result of this more focused purpose of the cores. This is further indicated by the fact that GPUs have specific cores like for example Tensor cores, RTX cores, and CUDA cores. All of these cores help to enable the GPU to be more performant at specific tasks, as the itself has a specific role within the system unlike the CPU which performs many different tasks and as such requires a more generally focused performant approach.

This can also be seen when discussing the amount of individual cores a GPU might have compared to a CPU. GPUs typically favour processing units over clock rate, whereas CPUs focus less processor units for more priority in the general performance of individual processors and therefore increased clock speeds for those processors. This means large scale tasks can utilise a lot more threads on the GPU, where the performing of small but large scale amounts of calculations are more easily performed. This all comes to result in a significant increase in performance as displayed in the two images shown above.