600086 Parallel and Concurrent Programming

ACW REPORT

By Russell Paine 531825

Word count: 1428

Introduction

This report will outline the design and implementation of the ACW for *600086 Parallel and Concurrent Programming.*

GPU Design

CUDA has been used to implement parralisation on the GPU with C++ as the containing language. Each kernel is designed to perform one task from the ACW specification, them being: Raycasting, Particle movement, and Gravity.

Raycasting kernel

This kernel is designed to turn a 3D environment into a 2D image through using raycasting. The screen size is how big the output image will be and for each of the pixels in that image a ray is fired from it. The kernel runs in parallel on a striped dataset so there is no need for locking on the variables, the amount of block and threads are designated by how big the output window is. The default is 512,512 which is broken down into 32,32 blocks with 16,16 threads. For each kernel run a ray is created from the eye position and it is checked if that ray intersects a particle in the 3D environment and if it is the pixel is set to the color of the particles it collided with. If it collides with multiple particles the closests is chosen.

Particles Movement

The amount of particles can be defined, the system can handle up to 10k well enough. The kernel runs with 100 threads per block and the amount of blocks is worked out by finding the number of particles divided by the threads per block. This performs the movement of each particle in parallel with each particle having its own thread. The Id is worked out by doing:

int i = blockIdx.x *blockDim.x + threadIdx.x;

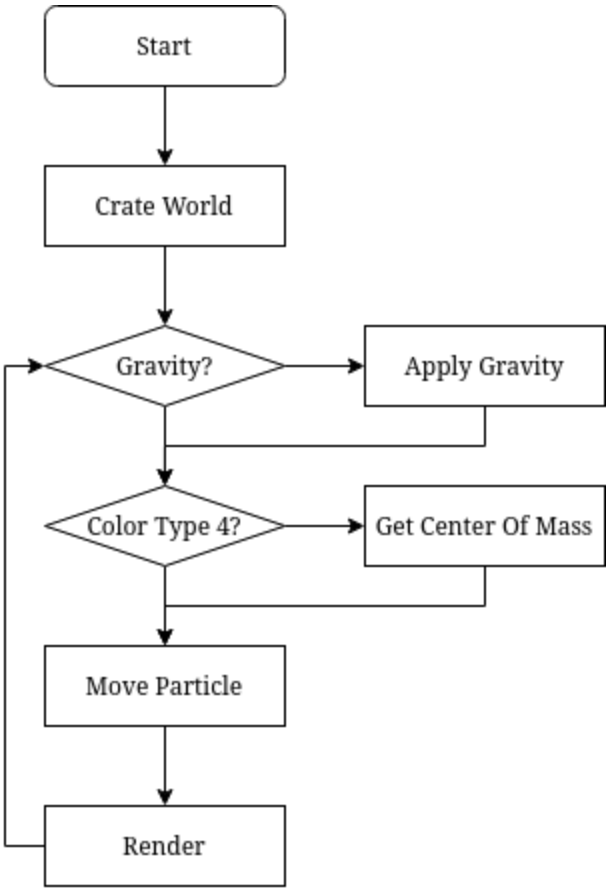
This get’s the unique id for the thread that is running and this allows the thread to know which particle it’s allowed to edit. The particle's position is then updated and if the color needs updating as well this is done here as well. The particles can be colored by either: solid color, brightness based on it’s speed, brightness based on the local force (amount of particles near it), and brightness based on distance from center of mass.

Gravity

Gravity is implemented similarly to the movement of particles. It gets the ID of the current thread then applies the movement. This is only done if gravity is enabled by the user.

Overall System Design

The overall flow of the the simulator is:



Crate world is a kernel that is one thread that creates all the spheres in the device memory so each kernel can receive a pointer to that list and utilise the particles

Check if gravity is required, if so apply the gravity.

Check if the color type is center of mass specific, if so this runs a one thread kernel to gather the average center of mass.

Move particles change their position based on their velocity, and update their color.

Render creates rays for each of the pixels of the screen and colors it on if they hit a particle.

The overall system didn't require much specific locking of variables, all of the kernels ran sequentially due to the fact that each step required the last to be completed before it could be run. This results in a nice loop that is done for every frame.

CPU Design

Python is used as the language to write the concurrent program. The CPU struggles a lot more concurrent tasks due to the hardware limitations. Python is also not known for it's amazing performance. The approach that was taken was to split each element that needs to be done into its own threads, instead of creating threads every time they are needed it was opted to create the threads once at compile time then have their function loop indefinitely until they are destroyed. This was chosen due to python having a large overhead in creating threads. This was achieved by giving each thread their own start and stop event. The threads that were created were: glut thread, gravity thread, update thread, and the raycast threads.

Overall System Design

The approach that was taken was to create a system that didn't require too much locking. The way this is done is by completing each task then allowing the next to run in the main thread. Due to the way the system runs there isn't much point in having all threads run at the same time due to the fact the render needs the whole update to run before it can. The use of mutual exclusion is important, if it wasn't implemented an unstable system would be the result. Using events as a building block to perform locking like operations on the data structures.

Glut Thread

This contains the glut loop and it started by an event, this renders the scene.

Gravity Thread

Applies the gravity force to each of the particles, there is only one so this thread runs for all the particles.

Update Threads

These threads run concurrently, they update a specific range of a striped list and change their position depending on its velocity.

Raycast Threads

The rays are created at compile time to save some CPU cycles, the thread has a range of values it needs to fire the ray for and check if it hits any particles, if it does it changes the color of the pixel it's currently running for.

Performance Comparison

It’s no surprise that the CPU blows the CPU out of the water in performance for parallel tasks. GPUs are designed for this sort of operation.

The CPU is running a grand total of 10 threads in its default form. For one thousand particles in the 3D environment it takes around 10 seconds to perform one frame. This is the whole system from start to end. Whereas the CUDA program is running over 1000 threads for the render alone. This means that it's able to cope with a lot more particles in the environment. Ten thousand particles can be rendered at about 20 frames a second on the testing system.

Python	FPS (Frame per second)	Amount of threads
16	3~4	10 threads
100	1	10 threads
1,000	0.1	10 threads
10,000	0.006	10 threads

CUDA	FPS (Frame per second)	Amount of total threads	Amount of render blocks + threads
16	60(Capped)	262210	8,2
100	60(Capped)	262346	10,10
1,000	60(Capped)	264146	100,10
10,000	20	282146	1000,10

Benchmark ran on Nvidia GTX 1070, Intel i5-6600k

It’s pretty clear that the GPU is amazing at performing tasks in parallel and pulling out huge performance gains.
An interesting thing to note is that python has a large performance drop in adding threads, it’s actually more performant to keep the program on a single thread. This is due to how python is handled behind the scene. With the testing setup with no threads the program can average about 2~3 frames a second without threading.

The conclusion to be drawn here is if a parallel task needs a solution; use a GPU.

Reflection

During this coursework a lot was learnt about CUDA, it's a technology that has a very high entrance knowledge requirement. There is a lot to learn to utilize CUDA to its full potential, this was definitely not achieved in this ACW. The way the particle simulation is implemented in the projects is an easier way, it could be more complex and be a more accurate simulation of particles in an enclosed space.

Lots have been learnt about how CPUs can gain performance from parallel programs but the way python is created it isn't true parallel and there isn't much performance gain, if any, from threading an application.

The basic system is all that was able to be implemented in the solutions, if other commitments didn't get in the way the advance system could have been implemented. A deeper understanding of particle system would need to be researched to achieve this.

Learning from this experience, doing this again substantial changes to the python solution would be done. The way python is implemented means that gravity and update could be running at the same time and lock the striped array if they both meet. A small performance gain could be seen.

The takeaway from this ACW; CUDA is extremely powerful but is a bigger time commitment to getting working and getting working well is even more. Python isn't a good option for parallel programs and the CPU has its limitations. The GPU is an amazing bit of hardware with its performance being mind boggling.