# University of Hull

# 

# Parallel particle systems Report

# Parallel and Concurrent Programming

# 

# 

**Maaz Selia**

| 3rd August 2020 |
| --- |
| Word count: ~1900 |

# CPU

The CPU implementation was based around a ***thread-safe queue***, with ***workers*** who receive and complete tasks from the queue. And ***managers*** who populate the queue with tasks. It also scales ***workers*** and ***workload*** based on the number of cores/threads on a system.

The ***thread-safe queue*** is a simple queue which allows for only 1 thread at a time to ***push*** or ***pop*** a task from the queue.

At initialisation, a ***systemManager*** thread is launched. This ***systemManager*** launches and manages sub-managers. These are: ***movementManager***, ***gravityManager***, and ***outputManager***. Before which, a ***workload*** limit is set, this limit is in relation to the number of particles and number of cores/threads in the system e.g. a ***workload*** of 10 is set, for 40 particles on a 4 thread system.

## Managers

These sub-managers launch ***x \**** ***workers***, where ***x = number of cores/threads***, and a local instance of a ***queue***. *[while running:]* they call sub-methods which ***push*** tasks onto the ***queue***. For ***workers*** to later ***pop***.

These sub-methods prepare tasks and ***push*** them onto the queue. Once all tasks have been pushed, ***x \* “Finished”*** messages are ***pushed*** to the ***queue***. (The ***x \**** is explained further below). Following this, a ***workAdded*** event is set and the sub-method goes into wait for a ***workCompleted*** event from the ***workers***. Then it is over to the ***workers***.

## Workers

When ***workers*** are launched. *[while running:]* they attempt to ***pop*** a task from the ***queue***. If the ***queue*** is empty, they go into waiting for a ***workAdded*** event from the ***manager***.

* When this event is set, the ***workers*** attempt to ***pop***. If a task is ***popped***, the task is completed, and the workers re-attempt a ***pop***.
* This is where the ***“Finished”*** messages are used. In order to ensure all ***workers*** have finished their tasks, all ***workers*** must have ***popped*** the ***“Finished”*** message.
* When a ***worker pops “Finished”***, a ***threadCounter*** is **atomically** incremented. If ***threadCounter*** <number of cores/threads, the ***worker*** goes into wait for a ***workersWakeUp*** event. So all ***workers*** besides the last go into waiting for this event.
* When the final ***worker*** completes its task and ***pops “Finished”***. It resets the ***threadCounter***, sets the ***workersWakeUp*** event and then sets the ***workCompleted*** event for the sub-method to continue.

For gravity, speed, force and center of mass. ***Flags*** are set in the form of events, and execution branches based on if ***flag.isSet()***.

That concludes the general overview. Following are the workings of the different ***managers***.

## movementManager

*[while running:]* if the ***fullMoveCompleted*** event is not set, ***moveParticle*** (sub-method) is called. The ***fullMoveCompleted*** event is set when ***moveParticle*** completes. And is cleared when ***outputManager*** generates an image.

***moveParticle*** distributes tasks in index ranges based on the ***workload*** limit. It distributes tasks in this way for movement, calculating force, and collision.

e.g. move particles from index 3 to 7, check collision for particles from index 7 to 11.

## gravityManager

This ***manager*** launches its ***workers*** and instantiates its ***queue*** and then simply waits for the ***killSystem*** event. The tasks for this ***queue*** are pushed from ***move-workers*** once a particle has moved an iteration. This allows for gravity to be applied in parallel with move.

## outputManager

*[while running:]* waits for the ***fullMoveCompleted*** event (from ***movementManager***), then acquires the ***outputLock*** and calls its sub-method. The sub-method creates ***100 \* 100 \* particles*** ray calculation tasks, which can all be performed in parallel. Once the sub-method is complete an ***outputReady*** event is set and the ***outputLock*** is released.

## OpenGL loop

This loop waits for the ***outputReady*** event, acquires the ***outputLock*** and updates the image. It then clears the ***outputReady*** event and releases the ***outputLock***. Allowing for a new image to be generated by the ***outputManager***.

## Miscellaneous

***keyboardLock***. This lock prevents a flag from being set mid way through a movement iteration. Acquired in ***keyboardCallback*** or movement sub-method.

***killSystem***. Initiated by pressing 5, allows for graceful termination of threads and exits the system.

# GPU

The GPU implementation was much simpler due to the intrinsic parallel nature of the hardware and language.

The ***render*** (host) method was the center point of the implementation. The simplified execution of this method is as follows:

| // 1 thread (particle, particle) pair  d\_collision <<< PARTICLE\_COUNT, PARTICLE\_COUNT>>>();  cudaDeviceSynchronize();    // 1 thread/particle  d\_moveParticles <<< 1, PARTICLE\_COUNT>>>();  cudaDeviceSynchronize();    if (gravity) {  // 1 thread/particle  d\_gravity <<< 1, PARTICLE\_COUNT>>>();  cudaDeviceSynchronize();  }    // 1 thread/pixel (100\*100)  d\_raycast <<< (4,4) , (25,25) >>>();  cudaDeviceSynchronize(); |
| --- |

## d\_collision

The ***collision kernel*** takes advantage of the block, thread system. Each block is taken to be the main particle and each thread, in that block, is taken to be a secondary particle.

This allows for each potential particle-particle collision to be checked on separate threads. The ***collision kernel*** has two ***\_\_shared\_\_*** variables: (bool) ***coll***; (vec3) ***finalNewVel***. These are initialised in the first thread, followed by a ***\_\_syncthreads()***. This is to ensure that the variables are initialized before the collision checks begin further below in the ***kernel***. Then each potential particle collision is checked and the ***shared variables*** are updated accordingly. Then before updating the particle velocities, another ***\_\_syncthreads()*** is placed, to ensure all collision checks have been completed before the velocities are updated.

## d\_moveParticles

The ***move kernel*** simply moves a particle followed by a ***\_\_syncthreads()***. This is to ensure all particles have been moved before any branching occurs based on colouring ***flags***.

One branch to note would be the ***if (Force)*** branch. If this ***flag*** is set, in thread 0, the move kernel dynamically calls a ***d\_force<<< 1, PARTICLE\_COUNT >>>()*** ***kernel***.

## 

## d\_gravity

A simple kernel which accelerates a particle. 1 thread per particle.

## d\_raycast

Adapted from the Ray tracing labs, this ***kernel*** is launched with a ***(4,4)(25,25)*** configuration. Therefore, 1 thread per pixel. However, the algorithm for raycasting could be reduced from ***O(n)*** to at least ***O(log n)*** if ***dynamic parallelism*** was to be employed within the ***kernel*** or grid depth increased based on ***PARTCILE\_COUNT***. Allowing for 1 thread per ray-particle collision detection.

## 

## Miscellaneous

***Colouring flags***. When a ***flag*** key is pressed, a ***setFlag (host)*** method is called. This method calls ***cudaDeviceSynchronize()*** to ensure a ***flag*** is not changed in the middle of a ***kernel*** execution. It then calls a ***setFlag kernel*** which sets a ***flag*** on the ***device*** accordingly. It then calls ***cudaDeviceSynchronize()*** again to make sure the flag is updated successfully before returning.

There is a major flaw in the GPU design, which would be the ***PARTICLE\_COUNT*** used as the input for thread configuration. Although it works, if ***PARTICLE\_COUNT*** was to go above the maximum threads available, the entire system would collapse. This was ignored for simplicity. Below is code which highlights a solution to the problem.

| int threadLimit = x;  int blocks = ceil(PARTICLE\_COUNT/threadLimit);  d\_example<<< blocks, threadLimit >>>(); |
| --- |
| d\_example {  Int index = threadIdx.x + blockDim.x \* blockIdx.x;  if (index < PARTICLE\_COUNT) {  // do stuff...  }  } |

# Performance

As expected, the GPU far outperformed the CPU. The GPU maintained ***O(1)*** performance from 5 particles to 50. And would continue to do so, as long as the number of particles was less than the thread/block limit. In areas where it fell short of O(1) performance, Force and Raycasting, it was due to a lacking parallel algorithm. In their implementations, both of these areas iterate through all particles once. If this iteration was also parallelised, the GPU would maintain, at least, ***O(log n)*** performance across all areas. *Tables below.*

The CPU was far behind the GPU in performance. I believe this was for two main reasons. The first being the GIL in python which restricts parallel processing of python byte-code. Which effectively meant that the CPU implementation, although a parallel solution, was running concurrently at best. The second being the massive difference in hardware threads between the CPU and GPU. Although the CPU had twice the clock speed, the GPU’s thread count left no room for the CPU to compete.

| GPU (1 iteration) | 5 particles | 50 particles |
| --- | --- | --- |
| Movement | ~0.05ms | ~0.05ms |
| Gravity | ~0.04ms | ~0.04ms |
| Force | ~0.04ms | ~0.12ms |
| Collision | ~0.1ms | ~0.1ms |
| Ray | ~4ms | ~28ms |

| CPU (1 iteration) | 5 particles | 50 particles |
| --- | --- | --- |
| Movement | ~0.35ms | ~3.5ms |
| Gravity | ~0.1ms | ~1ms |
| Force | ~0.15ms | ~1.5ms |
| Collision | ~0.15ms | ~1.5ms |
| Ray | ~500ms  (~0.01ms/ray) | ~5000 ms (~0.01ms/ray) |

The most noticeable difference between the two was observed in raycasting. Raycasting requires x(100 \* 100) ray-particle checks to be made, where x = number of particles. This heavily favoured the GPU with it’s massively parallel capacity, and cost major performance on the GIL-restricted CPU implementation.

The GPU seemed to favour tasks to be broken down into their smallest parts, for all to be processed in parallel without much overhead. Whereas the CPU seemed to favour when tasks were broken into groups and processed. As the overhead of managing threads, co-ordinating and handshaking when the processing was complete, ended up costing more than the performance gained from parallelising.

# 

# 

# Reflection

Parallelising is difficult. With serial code, not as much thought is needed for optimising your algorithms, as long as you avoid unnecessary loops and nested loops. You can be fairly confident that your code, though not extremely performant, will work well enough. Especially with the optimisations made by a compiler. However, if a solution was parallelised. An underdeveloped and unoptimised parallel algorithm could easily lead to the code becoming less performant than if the exact work was done serially.

To effectively optimise your algorithms. I believe, you need a deep understanding of the language and hardware and to be familiar with their intricacies. To know the performance cost of each line of code written, managing threads, distributing tasks, and handshaking. And to be able to optimise your algorithm specifically for the hardware.

As such, it takes a lot of knowledge and experience to code Parallelly effectively.

## CPU

For the CPU solution, I believe the in-built parallel scalability based on system cores/threads was its greatest strength. I believe the queue and worker system also worked well. To improve it further, I would look to create Template manager-queue-workers systems so that they can be easily deployed for parallelism of future solutions.

## GPU

Using the, almost ridiculously, large number of threads available on a GPU was a fun experience. Although managing the host and device separation was difficult. I believe my workaround to this, by simply doing most of the work on the device, was lazy and would not be feasible for other solutions. Learning to exploit both the host and device at the same time would be a valuable skill, especially with the performance focused nature of C/C++.

Parallelising was a lot easier compared to the CPU via the use of kernels, with the hardware being designed specifically for parallel processing and a language specifically for the hardware.

And although I made use of dynamic parallelism, I believe it was, unfortunately, ineffective usage. Learning to effectively use dynamic parallelism would be useful. As well as, a better understanding of the hardware. To be able to make use of techniques such as sequential addressing for reduction.

## Overall

In the future, I would look into ways and techniques to allow for better code debugging infrastructure and exhaustive testing of locks and synchronisations. As I believe, this was a lacking aspect during this project.

*End of report.*