N-body Algorithm Performance analysis

Sam Serrels
40082367@napier.ac.uk
Edinburgh Napier University
Concurrent and Parallel Systems (SET10108)

1 Introduction

N-body problems An N-body simulation is a system of elements, usually particles which interact in some way with every other element in the system. A common example of this would be particles with mass, which calculate the gravitational acceleration applied to them, other examples include plasma physics, fluid dynamics, and molecular dynamics. The main complexity of the simulation comes from the operation of each body interacting with every other body, causing a considerable amount of code execution. The Body-Body calculations, such as gravitational acceleration, could be a relatively simple piece of code, but will have to be executed N^2 times where N is the number of bodies. Therefore the complexity of an unoptimized simulation is simply $O(n^2)$

Project Scope This project investigates the performance of both a CPU and a GPU implemented N-Body Simulation. The CPU implementation was enhanced with the use of multi-threading with OpenMP. Although the initialization code and execution method for each of the two applications is vastly different, the simulation code is the exact same.

OpenMP The technology for processing the application in parallel was chosen to be OpenMp, an API that abstracts the creation of threads from the user and therefore allows for easier development and better cross platform portability, assuming that the chosen platform has a complier that supports OpenMP. This was chosen over creating threads manually, mainly for ease of development reasons, but also because even in a situation that OpenMp is slower than Manual threads, there should still be a noticeable performance increase over the baseline results.

Simulation Implementation A simplified version of a gravitational simulation was used, each particle has a mass of 1. A further method was added to attract all bodies back to the center. This simulation was chosen for visual appeal, as only the computational time was being measured.

2 CPU implementation

Analysis of the cpu code revealed expected results, the main simulation loop took up almost 100% of the execution time. Figure 1 visualizes the main loop and highlights the lines of code which take the most time.

3 CPU Results

Different versions of the program were run using a system containing an Intel i7-4790K cpu at 4GHz. Only the time taken to run a step(or "Tick")of the simualtion was measured, time taken to render the particles was not measured. Each result is a average of 1000 ticks where possible, with high particle counts the sample size was gradually reduced to 10 samples.

3.1 System utilization

Shown in Figure 13 and Figure 14. As the mathematical workload of the simulation is straightforward, the cpu utilization was an expected uniform value, corresponding to the amount of threads. With 8 threads, cpu utilization was between 95% and 100%. With anything less, the results were almost exactly equal to the ratio of threads to maximum logical cores (4 threads: 50%, 1 thread: 12.5%, 2 threads: 25%).

```
        for (int i = 0; i < PARTICLESIZE; i++) {
          vec3 newVelo(0, 0, 0);
 1.7 %    for (int j = 0; j < PARTICLESIZE; j++) {
 9.0 %      vec3 r = bodies[j].pos - bodies[i].pos;
11.6 %      float distSqr = dot(r, r) + SOFTENING;
 4.1 %      if (distSqr > 0.1f) {
 5.5 %        float invDist = 1.0f / sqrtf(distSqr);
11.5 %        float invDist3 = invDist * invDist * invDist;

54.6 %        newVelo += r * invDist3;
            }
          }

<0.1 %    bodies[i].speed += delta * newVelo * FRICITON;
<0.1 %    bodies[i].speed -= 0.000001f * bodies[i].pos;
<0.1 %    bodies[i].pos += bodies[i].speed;

<0.1 %    bodies[i].pos.x = clamp(bodies[i].pos.x,-1000.0f,1000.0f);
<0.1 %    bodies[i].pos.y = clamp(bodies[i].pos.y,-1000.0f,1000.0f);
<0.1 %    bodies[i].pos.z = clamp(bodies[i].pos.z,-1000.0f,1000.0f);
        }
```

Figure 1: ***Main Simulation Loop*** *- Measure of execution Time*

3.2 Core occupancy

Shown in Figure 10, Figure 11, and Figure 12. The threads used in the program jumped between logical cores frequently, even when running the program for long periods of time with large particle counts. It is not clear how much of a performance hit these transfers have caused on the application. Changing OpenMP scheduling parameters did not seem to produce a noticeable difference to this.

3.3 Speedup

Shown in Figure 6 and Figure 8 for above 2048 particles, speedup is roughly ranked by threads count, with 8 threads producing the highest speedup. A thread count of 4 produced a fluctuating speedup, coming close to beating 6 threads at times. This could be due to the CPU having 4 physical cores, which are hyper-threaded to 8 logical cores.

3.4 Efficiency

Shown in Figure 7 and Figure 9. While running with 2 threads produced the least amount of speedup, it was far ahead of any other thread count in terms of efficiency. This was true for all but the largest particle count of 524288, where both 3 and 4 threads were more efficient and than 2. This is most probably an artifact of the CPU/Operating System thread scheduling when put under heavy usage.

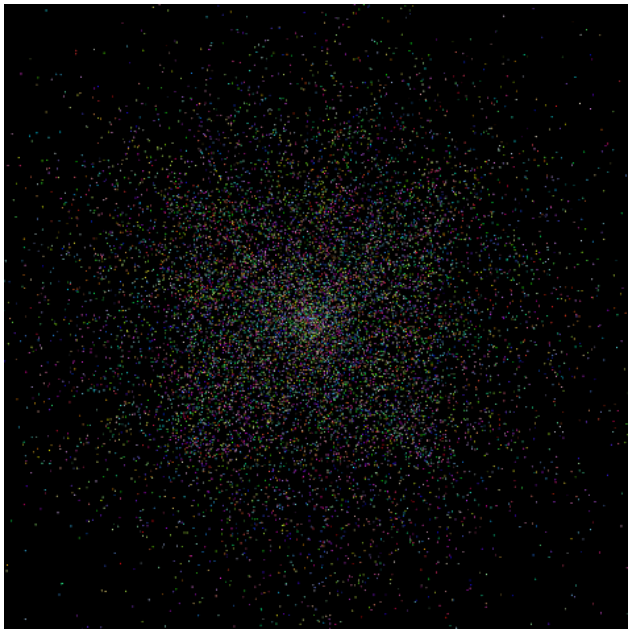

Figure 2: ***Cpu Simulation*** *- 8096 Particles*

4 GPU implementation

The GPU version of the application is written as a DirectX12 Compute Shader, the output of the computation was passed to a geometry shader which turned the simulation body data into triangles which could then rendered. This method keeps all the work on the GPU with the CPU only used for synchronization. As the velocity data is already on the GPU and accessible by the renderer, it was used to colour the particles, faster particles are rendered with a brighter colour.

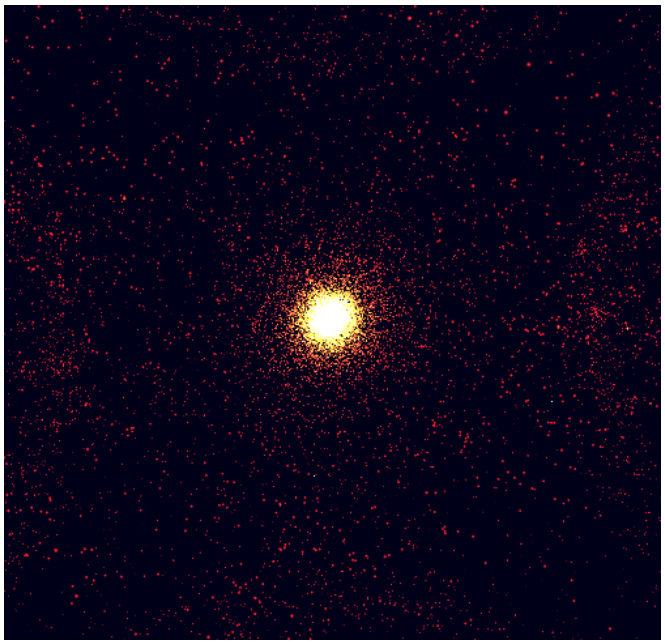

Figure 3: ***Gpu Simulation*** *- 65536 Particles*

4.1 GPU Computation time

The hardware used was a Nvidia GTX 980, which has 16 Multiprocessors, with each with 128 Streaming processors (2048 total cores). The GPU is well suited for this type of application, as the results show, the GPU is between 15% and 160% faster than a cpu running with 8 threads. The GPU kept below 16ms (required for 60fps) computation time for up to 32768 particles (The CPU could only do 2048, using 8 threads).

4.2 Work Group Size

Work is dispatched to the GPU in 3 Dimensional groups of kernels. A group of kernels should theoretically all run at the same time, as N-Body requires no intercommunication or synchronizing between kernels, the group size could be set to any size supported by the hardware. This report measures the effect of varying the work group size (Figure 15 and Figure 16).
Using DirectX 12 a group dimension cannot exceed 65535, hence the missing value for (16,524288) in the results.
Analyzing the data, it seems that a group size of 32 consistently provides the quickest computation time. This could be due to 32 also being the "Warp Size" of the hardware, which is the amount of instructions that are issued at once from a multiprocessor

5 Future work

5.1 Optimization algorithms

There are existing algorithms to reduce the problem size by using varied forms of approximation.

By grouping the Bodies into discreet areas (normally a tree structure), groups of bodies can be averaged to one central point for an area. When looping through each body, calculations may only use the averaged position of a collection of bodies, if the area they reside in is far enough away. This approach theoretically reduced the complexity from $O(n^2)$ to $O(n \log n)$ [Barnes and Hut 1986]

Implementing the method for creating and traversing a tree of bodies is relatively straightforward on the CPU and can have large performance gains. Implementing the same thing on a GPU would be a harder task [Jiang and Deng 2010]. Parallelizing the tree creation stage would require frequent synchronization, as the dimensions and state of the tree change for every body that is added to it.

6 Conclusions

This report evaluated the performance of two similar implantations of an N-Body particle simulation. The parallel performance of a CPU was compared to that of the GPU and it was found that the GPU is significantly faster at this task. Using threads on the CPU was found to be worthwhile, but the upper limit of performance was still significantly behind that of the GPU. N-Body simulations are very easily converted from sequential to parallel, due to there being no dependencies between each bodies calculations, therefore this type if simulation provides a good benchmark of total available compute resources.

References

BARNES, J., AND HUT, P. 1986. A hierarchical O(N log N) force-calculation algorithm. *324* (Dec.), 446–449.

JIANG, H., AND DENG, Q. 2010. Barnes-hut treecode on gpu. In *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on*, vol. 2, 974–978.

7 Appendix

Test Name	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288
GTX980 512	0.233	0.321	0.652	1.145	1.974	9.219	36.789	135.558	500.509	2100.374
GTX980 256	0.222	0.307	0.612	1.065	1.937	9.162	34.392	134.494	489.240	2051.084
GTX980 128	0.132	0.194	0.296	0.716	1.891	8.721	53.158	120.636	471.306	1902.573
GTX980 64	0.132	0.172	0.277	0.645	1.862	7.259	31.745	118.255	451.687	1901.846
GTX980 32	0.110	0.157	0.289	0.640	1.833	7.255	28.108	116.337	451.348	1899.133
GTX980 16	0.150	0.224	0.464	1.468	3.561	13.522	63.056	222.645	917.328	0.000
CPU 1 core	4.898	19.137	74.248	296.022	1185.570	4798.358	19412.055	76464.740	314798.033	1227484.000
CPU 2 cores	2.510	9.825	39.221	155.133	613.820	2451.838	9838.845	40324.536	167580.294	797311.542
CPU 3 cores	2.180	9.193	31.041	119.769	455.580	1821.591	7185.361	27788.974	115910.695	496237.950
CPU 4 cores	1.674	7.044	22.304	98.299	410.325	1597.991	5535.962	21020.147	95596.342	377586.191
CPU 5 cores	2.018	6.930	25.998	89.877	375.376	1455.549	5624.133	22419.623	92536.925	366961.358
CPU 6 cores	1.547	6.081	23.429	89.602	350.761	1350.470	5229.090	20683.325	84305.466	350425.495
CPU 7 cores	1.538	5.044	20.425	82.377	322.066	1282.543	4918.743	19273.623	78993.126	325729.737
CPU 8 cores	1.674	5.106	18.330	72.642	285.565	1178.308	4643.889	18259.532	72826.053	302954.399

Table 1: *Results of all tests, Times in Milliseconds*

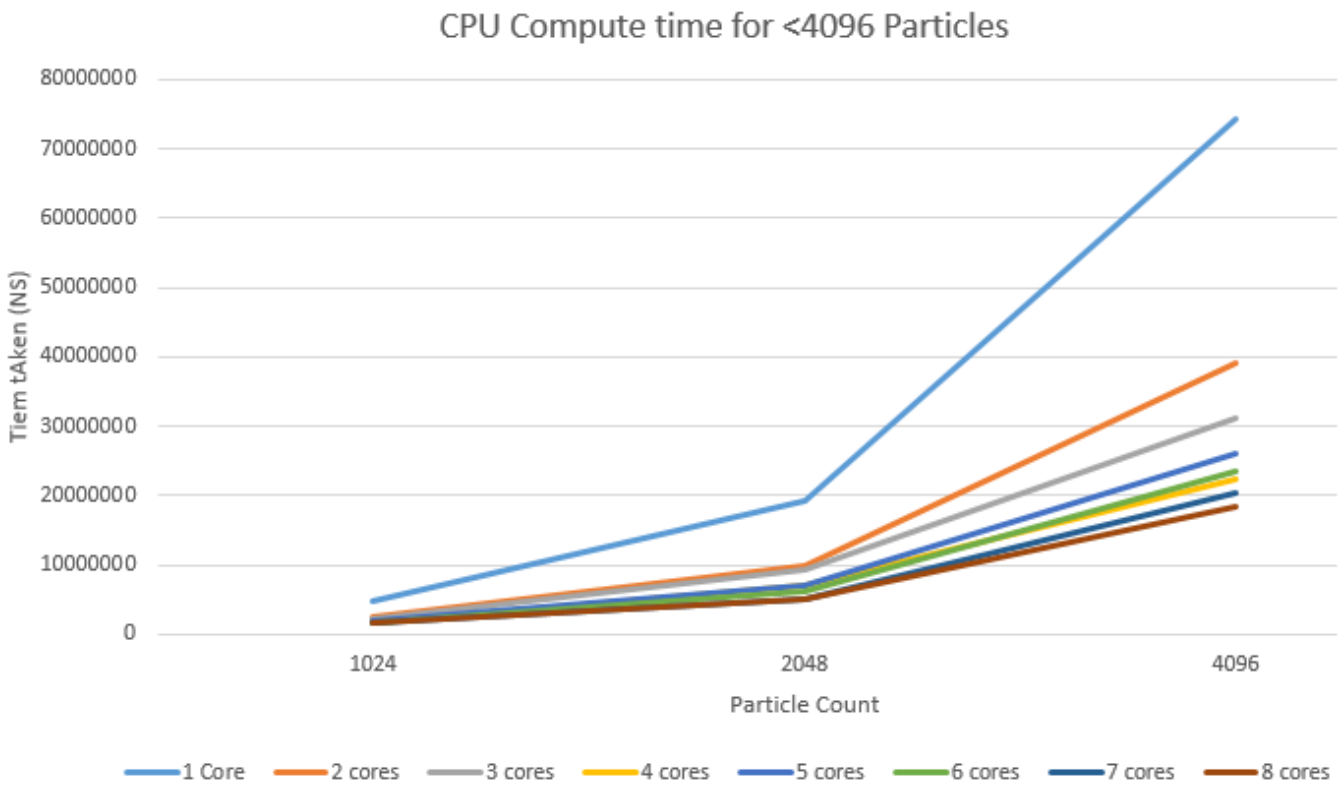

Figure 4: ***Cpu Compute time for less than 4096 Particles*** *- Time Measured in Nanoseconds*

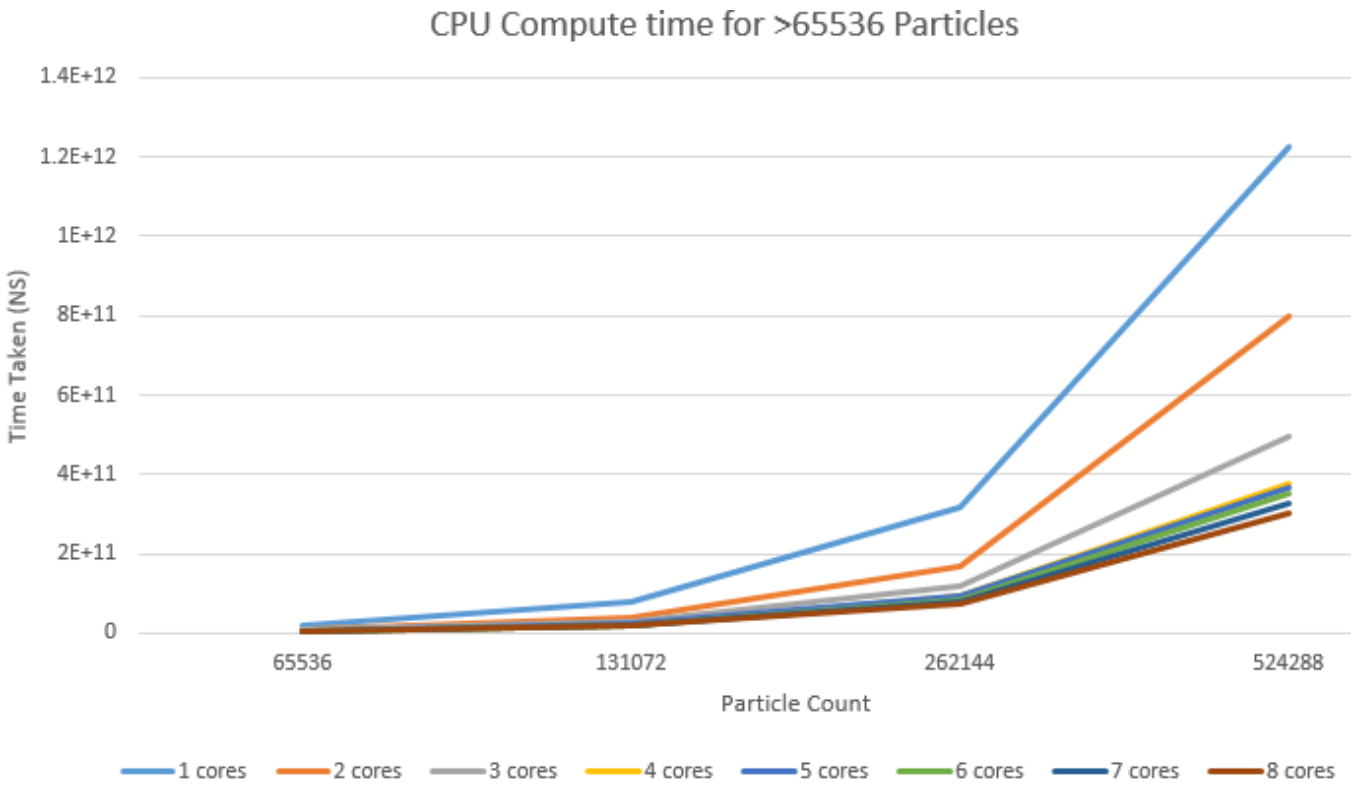

Figure 5: ***Cpu Compute time for greater than 65536 Particles*** *- Time Measured in Nanoseconds*

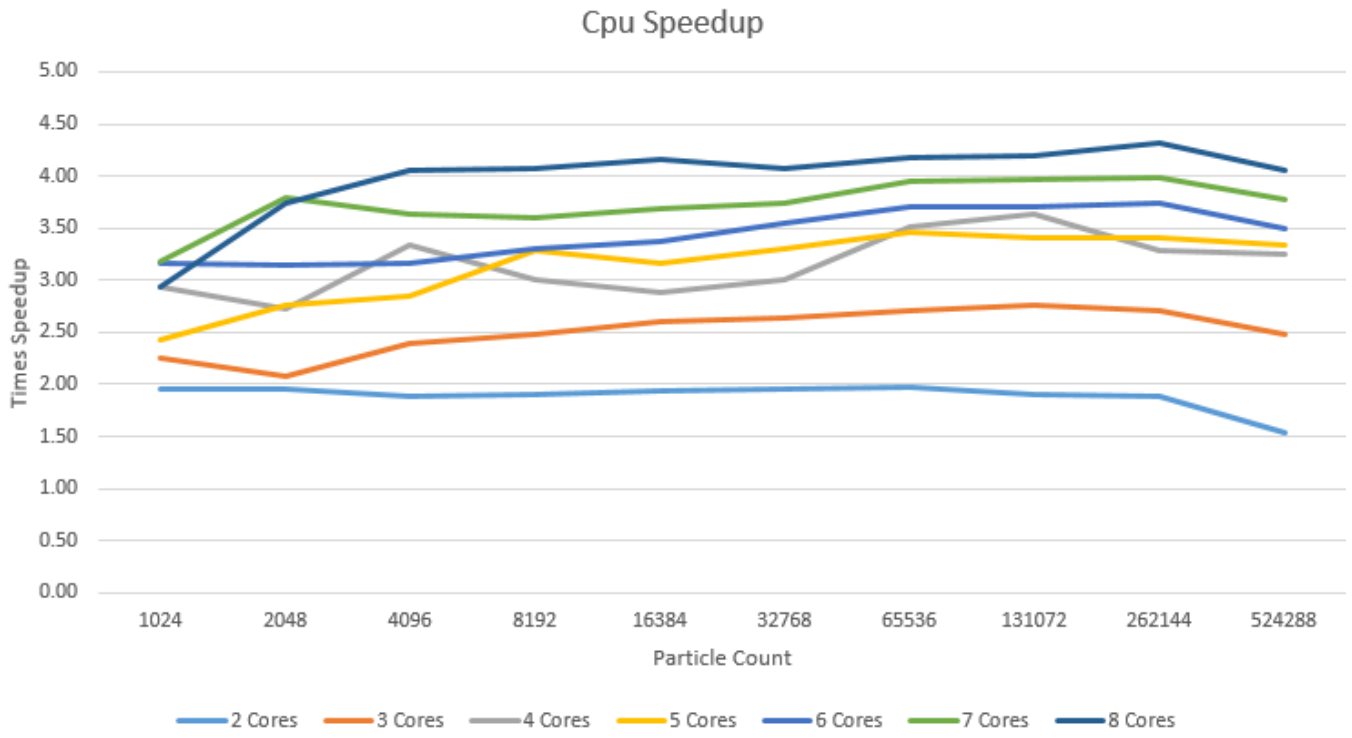

Figure 6: ***Cpu Speedup*** -

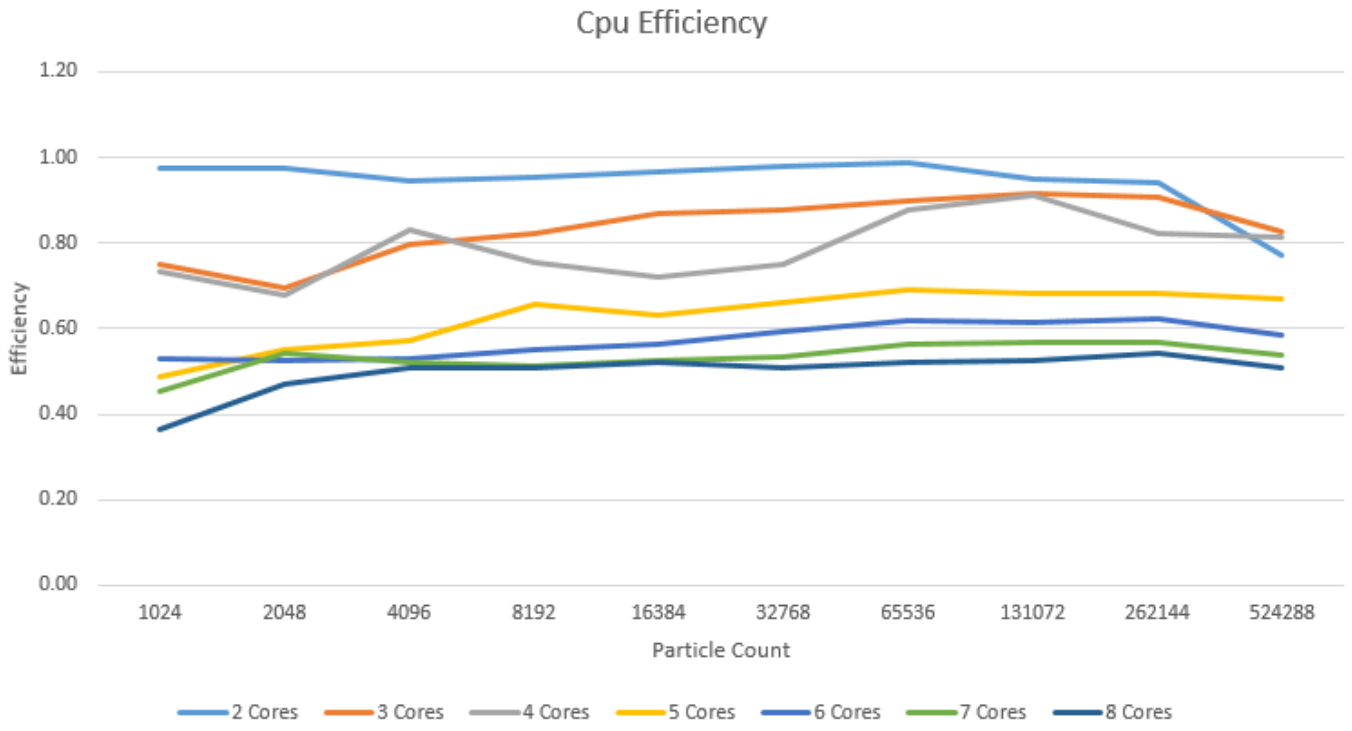

Figure 7: ***Cpu Efficiency*** - *Speedup measured against core count*

Speedup										
	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288
2 Cores	1.95	1.95	1.89	1.91	1.93	1.96	1.97	1.90	1.88	1.54
3 Cores	2.25	2.08	2.39	2.47	2.60	2.63	2.70	2.75	2.72	2.47
4 Cores	2.93	2.72	3.33	3.01	2.89	3.00	3.51	3.64	3.29	3.25
5 Cores	2.43	2.76	2.86	3.29	3.16	3.30	3.45	3.41	3.40	3.34
6 Cores	3.17	3.15	3.17	3.30	3.38	3.55	3.71	3.70	3.73	3.50
7 Cores	3.18	3.79	3.64	3.59	3.68	3.74	3.95	3.97	3.99	3.77
8 Cores	2.93	3.75	4.05	4.08	4.15	4.07	4.18	4.19	4.32	4.05

Figure 8: ***Cpu Speedup*** *- Colour coded scale for visual clarity*

Efficiency										
	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288
2 Cores	0.98	0.97	0.95	0.95	0.97	0.98	0.99	0.95	0.94	0.77
3 Cores	0.75	0.69	0.80	0.82	0.87	0.88	0.90	0.92	0.91	0.82
4 Cores	0.73	0.68	0.83	0.75	0.72	0.75	0.88	0.91	0.82	0.81
5 Cores	0.49	0.55	0.57	0.66	0.63	0.66	0.69	0.68	0.68	0.67
6 Cores	0.53	0.52	0.53	0.55	0.56	0.59	0.62	0.62	0.62	0.58
7 Cores	0.45	0.54	0.52	0.51	0.53	0.53	0.56	0.57	0.57	0.54
8 Cores	0.37	0.47	0.51	0.51	0.52	0.51	0.52	0.52	0.54	0.51

Figure 9: ***Cpu Efficiency*** *- Speedup measured against core count*

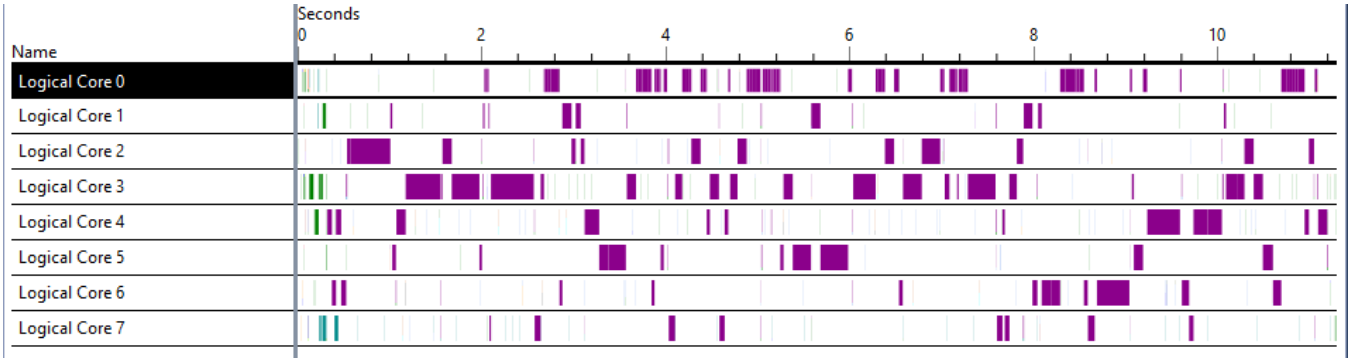

Figure 10: ***Cpu Occupancy*** *- 1 Core, 4096 Particles*

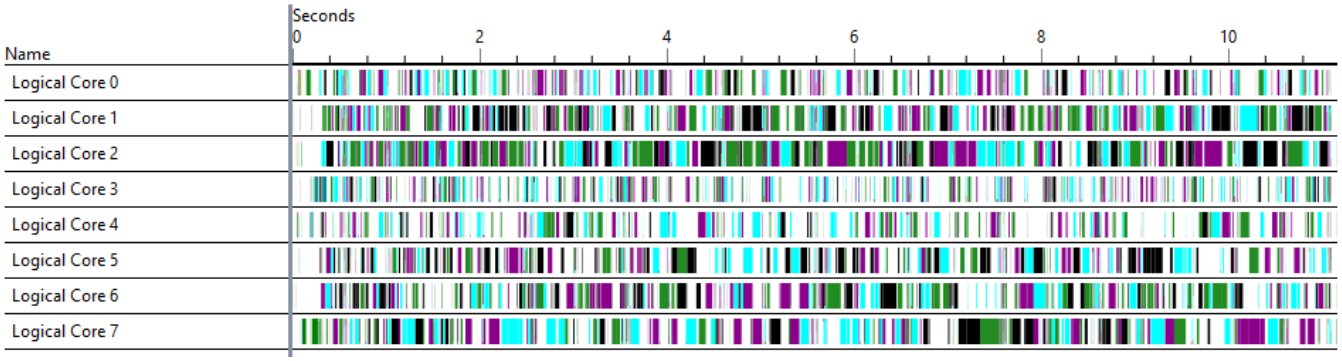

Figure 11: ***Cpu Occupancy*** *- 4 Cores, 4096 Particles*

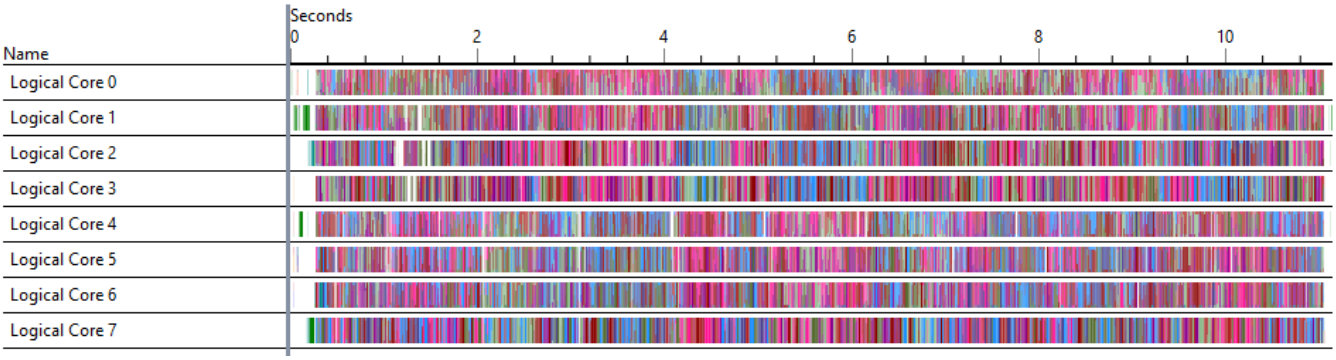

Figure 12: ***Cpu Occupancy*** *- 8 Cores, 4096 Particles*

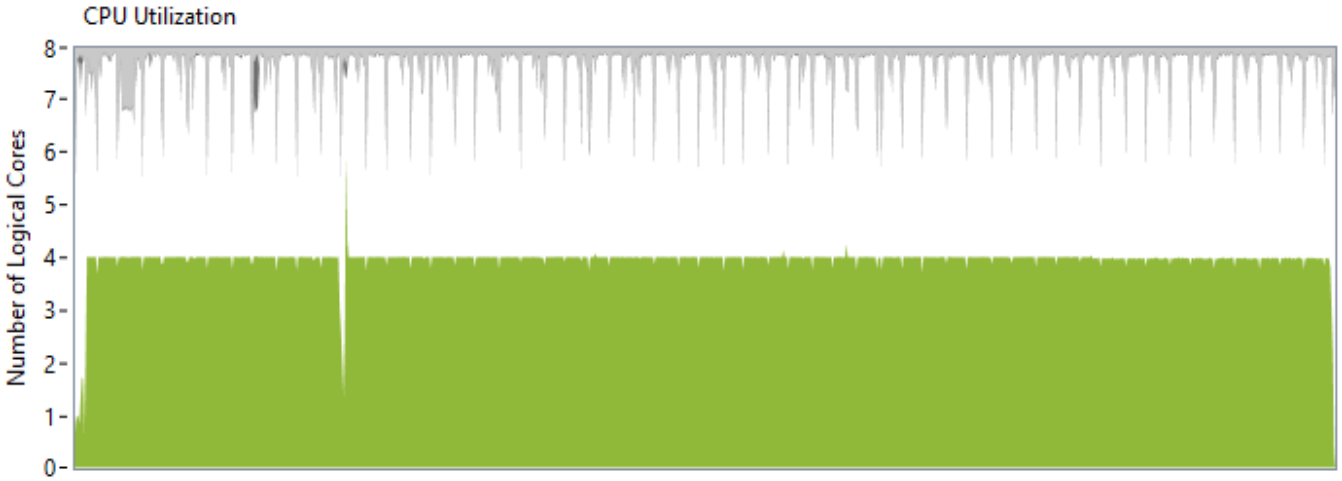

Figure 13: ***Cpu Utilization*** *- 4 Cores, 65536 Particles*

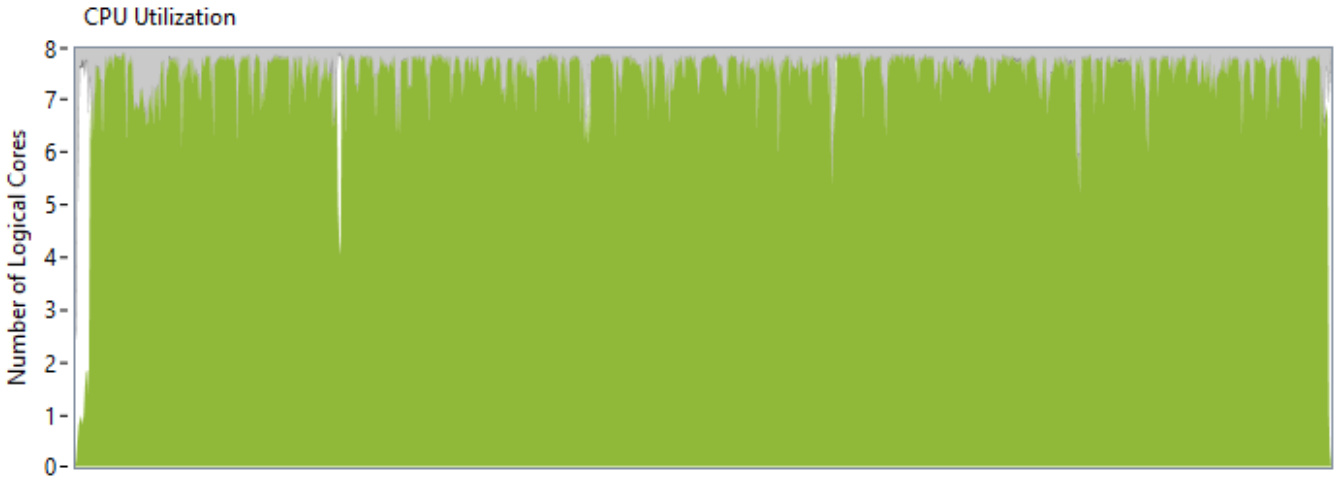

Figure 14: ***Cpu Utilization*** *- 8 Cores, 65536 Particles*

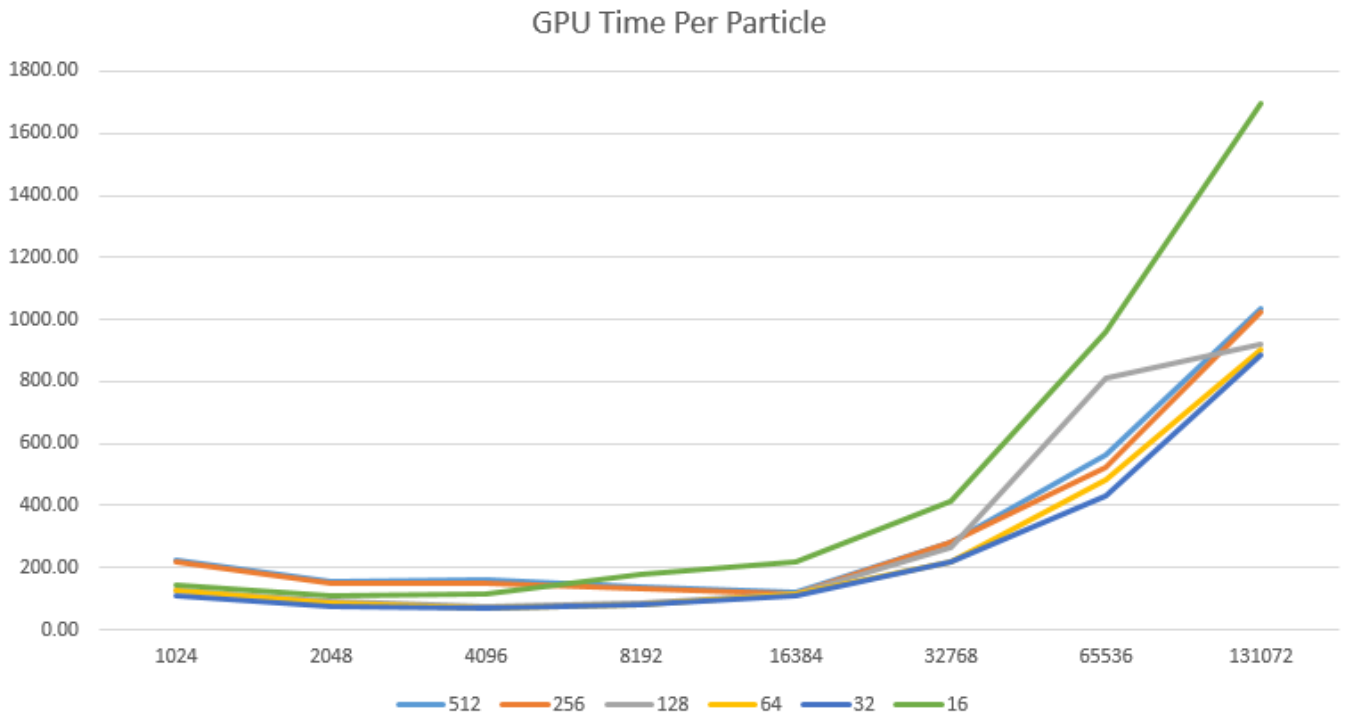

Figure 15: ***GpuTme Per Particle*** *- Varying Workgroup size*

GPU Time Per Particle

	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288
512	227.11	156.97	159.22	139.77	120.49	281.33	561.35	1034.23	1909.29	4006.14
256	217.17	149.79	149.37	129.97	118.21	279.62	524.79	1026.11	1866.30	3912.13
128	128.92	94.75	72.25	87.36	115.39	266.14	811.13	920.38	1797.89	3628.87
64	128.66	84.20	67.60	78.80	113.67	221.52	484.38	902.21	1723.05	3627.48
32	107.18	76.43	70.64	78.17	111.85	221.42	428.89	887.58	1721.75	3622.31
16	146.61	109.34	113.19	179.17	217.37	412.66	962.15	1698.65	3499.33	

Figure 16: ***GpuTme Per Particle*** *- Varying Workgroup size*