SET10108 Concurrent and Parallel Systems Report for Coursework Part 2

Beej Persson, 40183743@live.napier.ac.uk
School of Computing, Edinburgh Napier University, Edinburgh

For part 2 of the coursework required for the SET10108 Concurrent and Parallel Systems module at Edinburgh Napier University an *n*-body simulation application's performance was to be evaluated and improved by utilising parallel techniques. This report documents one such investigation where the algorithm was parallelised and the difference in its performance was measured.

***Index Terms*—parallel, *n*-body, OpenMP, CUDA, C++11, performance, speedup, efficiency.**

I. Introduction and Background

THE aim of this report is to evaluate the performance of an n-body application and attempt to improve this performance using parallel techniques. The n-body algorithm used initially processes sequentially on a single core of the CPU, but by investigating different parallel techniques the algorithm was changed to run on either multiple CPU cores or the GPU in an attempt to increase the performance.

A. N-body Problem

The N-body problem is the problem of attempting to predict the positions and velocities of a group of bodies whilst they interact with each other via gravity. Finding a solution to this problem is generally done by calculating the sum of the forces acting on a each body in the system and using this to estimate its velocity and position. Given a body's mass, m_i, and position, p_i, the force acting on it by another body, m_j and p_i is given by Equation 1 below (Reference [1]):

$$F_{ij} = \frac{Gm_im_j(p_j - p_i)}{||p_j - p_i||^3} \tag{1}$$

Where F is the force and G is the gravitational constant. Using this equation, and knowing that $F = ma$, the acceleration of the body can be determined. Thus the new velocity and position can be found by multiplying the acceleration by a chosen timestep. To produce an n-body simulation this calculation can be done multiple times with a small enough timestep to accurately model the movement of the bodies in a space.

B. N-body Simulation

The application used to generate an n-body simulation was written in C++ using a combination of two n-body algorithms available online. The structure of the application was based on Mark Harris' [2], whilst much of the maths used to calculate forces was based on Mark Lewis' [3]. To ensure the algorithm operated as intended, the simulation was visualised by generating a data file of the body's positions and radii at each timestep and running a python script that converted that data into a video file.

II. Initial Analysis

Upon running the application a few times and changing the number of bodies and the number of iterations of the simulation, an idea of its baseline performance was gathered. Below, in Tables I and II, the results of this initial testing on the sequential algorithm can be seen.

TABLE I
1000 Iteration Sequential Algorithm Performance

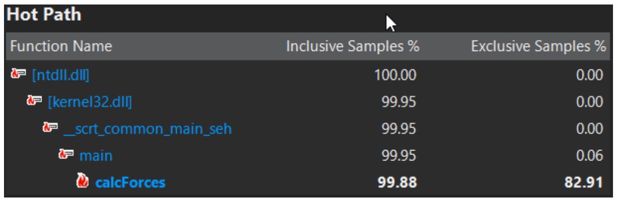
Simulation Iterations = 1000	
Number of Bodies	Average Time / ms
64	20.3
128	80.87
256	322.96
512	1289.07
1024	5119.45
2048	20196.89
4096	80797.32

The application's $calcForces$ method had complexity of O(n^2): it contained a nested forloop, where each body in the system was compared against every other body. Given this, increasing the number of bodies results in the time taken to run 1000 iterations of the simulation to increase at an n^2 rate, as seen in Table I. However, running more simulation iterations resulted in a linear increase in the time taken to simulate 1024 bodies, with almost perfect positive correlation, as can be seen in Table II.

TABLE II
1024 bodies Sequential Algorithm Performance

Number of Bodies = 1024	
Simulation Iterations	Average Time / ms
250	1337.40
500	2699.15
750	3884.60
1000	5214.85
1250	6523.30
1500	7799.55

Further to this, a performance profiler was run to identify the possible bottlenecks and determine the best areas to attempt parallelisation to improve the application's performance. As can be seen in the code's hot path in Figure 1, the *calcForces* method, discussed earlier, was what took up the majority of processing time when there was a significant number of bodies.


Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
[ntdll.dll]	100.00	0.00
[kernel32.dll]	99.95	0.00
__scrt_common_main_seh	99.95	0.00
main	99.95	0.06
calcForces	**99.88**	**82.91**

Fig. 1. A image showing the results of running Visual Studio's Performance Profiler, where the "Hot Path" is displayed.

This method, therefore, was the area that was parallelised in attempt to improve performance. However, when there was only 2 bodies the method executed very quickly showing that each individual comparison between bodies required little processing time. The high processing time when there was a large number of bodies was simply that there were so many comparisons to be made. This opened up GPU parallelisation as a potential solution given the large number of lower clock-speed cores.

III. Methodology

A systematic approach was undertaken to evaluate the performance of the algorithm and to measure any improvements in performance gained by parallelising the algorithm. The parallelising technologies were chosen based on the results of the initial analysis and on their ability to maximise the performance of the application.

The first step was to run a series of tests on the application to determine likely areas that could be parallelised and which technologies would be suitable, and to provide a baseline that the performance of the different parallel implementations could be compared to. These tests were all done on the same hardware, the relevant specifications of which are shown in table III. The details of the tests are shown in the testing subsection below.

TABLE III
Hardware Specifications

CPU	i7-4790K 4 Core HT @ 4.00GHz
GPU	NVIDIA GTX 980 @ 1.12MHz
RAM	16.0GB
OS	Windows 10 Pro 64-bit

A. Parallelisation Techniques

After these benchmarks for the sequential algorithm were recorded, the chosen parallelising techniques were applied to the algorithm and some preliminary simulations were run, each time checking the visualised output of the application. The intention here was twofold; to ensure that the techniques had been implemented correctly, that the parallelised algorithm was still producing the same simulation as the sequential application, and to gain an idea of their relative performance. The techniques used were OpenMP and CUDA, and the parallelisation was only applied to the $calcForces$ method as it took the majority of processing time and in an attempt to reduce accidental speedup to the application beyond simply parallelising the sequential algorithm itself.

1) OpenMP

OpenMP is an API that supports shared-memory parallel programming and allows some additional manipulations in the scheduling that were used in an attempt to increase performance. The pre-processor argument shown in Listing 1 was used to parallelise the outer for-loop, allowing the force calculation algorithm to be run across multiple threads.

```
void calcForces(Body *p, int numBodies) {
#pragma omp parallel for schedule(static)
  for (int i = 0; i < numBodies; ++i)
  {
    /* nested forloop force calculations */
  }
```

Listing 1. The OpenMP parallel for used to parallelise the shown forloop across the number of threads desired.

OpenMP's parallel for function comes with a *schedule* clause, seen in Listing 1, that was set to static as each iteration of the loop took the same amount of time to process. OpenMP was chosen as a parallel technique as the initial analysis showed that the $calcForces$ method was taking a long time to compute when run sequentially, and parallelising this algorithm so that it could run on multiple threads simultaneously would improve the performance of the application. OpenMP was chosen as it provides this parallelisation simply and effectively across the available CPU cores.

2) *CUDA*

CUDA is an API and parallel computing platform created by NVIDIA that allows software to utilise a CUDA-enabled GPU's virtual instruction set and execute the application in parallel using compute kernels. CUDA allows the user to determine the number of *blocks* and *threads per block*, showing in Listing 2, to be used for the kernel method and some testing was done to determine the optimal ratios. Below is an excerpt from the CUDA application.

```
__global__ void calcForces(Body *p, int n) {
  int i = (blockDim.x * blockIdx.x) + threadIdx.x;
  if (i < n) {
    /* nested forloop force calculations */
  }
int main() {
  ...
    calcForces<<<BLOCKS,THREADS_PER_BLOCK↩
      >>>(d_p, numBodies);
  ...
}
```

Listing 2. The $calcForces$ method with CUDA adjustments made and how the kernel method is called within the $main$.

CUDA was chosen as a notable contrast to OpenMP as it runs on the GPU and would enable a comparison between their respective performance. Further to this, given the results of the initial analysis, executing the $calcForces$ method in parallel on the GPU would provide significant speedup, especially for larger numbers of particles, due to the large number of available cores.

B. *Testing*

The same series of tests that were run on the sequential algorithm in the initial analysis were then undertaken for each implemented parallelisation. These tests were done under the same conditions and on the same hardware to eliminate discrepancies. For the majority of the OpenMP tests, the algorithm was run on the maximum number of threads available. CUDA allowed effective cores in use to be determined as attributes in the kernel method, therefore some initial testing was done on the performance of the application with the number of *threads per block* manipulated to identify the number of *blocks* and *threads per block* that would be used in the later tests.

The testing parameters used to evaluate the performance of the algorithms and the tests themselves are listed below.

For all tests the dependent variable being measured was the amount of time it took for the applications to produce the required number of iterations of the simulation. For each change in the independent variables, 100 tests were run and the time it took for the algorithm to produce the necessary values recorded, before the average run times were calculated.

1) CUDA's Threads Per Block
   - Independent variables: threads per block (from 1 to 32 incremented by powers of 2).
   - Constants: simulation iterations (1000), number of bodies (1024).
2) CUDA and OpenMP Comparisons
   - Independent variables: number of bodies (from 64 to 4096 incremented by powers of 2)
   - Shared constants: simulation iterations (1000).
   - CUDA constants: threads per block (8), blocks (number of bodies divided by threads per block).
   - OpenMP constants: threads (8).

3) Single Thread Tests

- Independent variables: each algorithm (sequential, CUDA, OpenMP).
- Constants: simulation iterations (1000), number of bodies (512), threads (1).

C. Evaluation

The results of these tests were then collated and compared to the results from the sequential algorithm's testing and used as the basis for the evaluation of their respective performance.

To represent the improved performance, the speedup, S, and efficiency, E, of the algorithms was calculated using the formula shown in Equation 2 below:

$$S = \frac{T_{serial}}{T_{parallel}}, \qquad E = \frac{S}{p} \tag{2}$$

Where T_{serial} and $T_{parallel}$ are the sequential and parallel computational times respectively, and p is the number of processor cores. When calculating the efficiency of the OpenMP application, p was set to 4, which was the number of available CPU cores on the test hardware. However, determining p for the CUDA application was more complex. Whilst the GPU used to test the application was listed as having 2048 cores, that wasn't necessarily the number of hardware cores assigned when running the kernel method. Instead, p was calculated as the number of *blocks* multiplied by the number of *threads per block* set as attributes in the kernel method, as this results in the total number of threads doing work on the GPU.

IV. Results and Discussion

The results of the initial test on the CUDA application can be found in Table IV below.

TABLE IV
CUDA Performance: Threads Per Block

Number of Bodies = 1024, Simulation Iterations = 1000	
Threads Per Block	Average Time / ms
1	3126.17
2	1183.69
4	1051.98
8	1058.97
16	1117.77
32	1256.76

When the kernel method was called with only a single *thread per block*, the application took significantly longer to generate a 1000 iterations of the simulation of 1024 bodies than when using a higher number of *threads per block*. Whilst there was a noticeable increase in performance when run using 2 *threads per block*, the average time seemed to reach its lowest at around 4 or 8 *threads per block*, before again increasing at 16 or more *threads per block*. These results are visualised in Figure 2 below. The error bars on this graph were determined by calculating the standard deviations of each set of 100 test results, which, as can be seen, were very small.

Given these results, 8 *threads per block* were used for all CUDA performance tests undertaken afterwards, and *blocks* were set to the number of bodies being tested divided by *threads per block*, to ensure each body would receive its own thread, maximising use of the available hardware.

The results from the performance testing done on the applications can be seen summarised in Table V below.

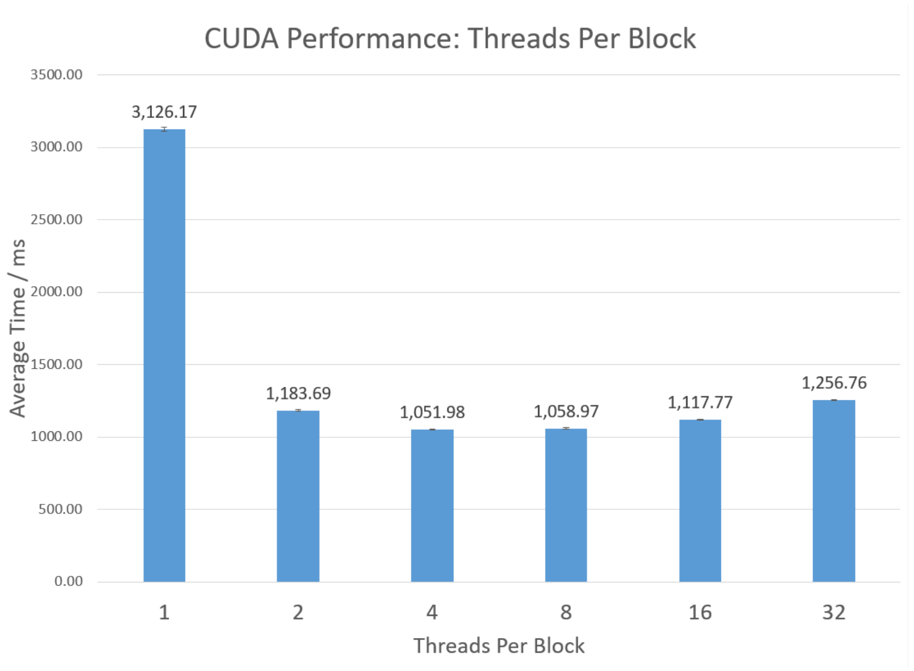


Fig. 2. A graph showing the average time it took the CUDA application to generate the simulation data when changing the allocated number of threads per block on the GPU.

TABLE V
1000 ITERATIONS N-BODY SIMULATION

Number of Iterations = 1000			
Application	Sequential	OpenMP	CUDA
Number of Bodies	Average Time / ms		
64	20.39	15.46	105.75
128	80.87	45.81	156.79
256	322.96	148.66	285.48
512	1289.07	533.66	528.34
1024	5119.45	2007.81	1063.36
2048	20196.89	7820.01	2398.78
4096	80797.32	30628.95	6822.16

As discussed in the initial analysis there is an n^2 positive correlation between the average time and the increasing number of bodies for the sequential application, but this also holds true for the OpenMP application. For the CUDA application, however, the relationship is more complicated. For a small number of bodies, the average time taken to generate an n-body simulation was higher than both the OpenMP and sequential applications. But as the number of bodies became increasingly large, the average times became significantly lower than the other applications. For both parallel applications, whenever there was a more significant number of bodies being simulated, the average times in which they produced the data were much lower than the sequential application.

Below, in Figure 3, is the graph of these results. The error bars on this graph were once again determined by calculating the standard deviations of each set of 100 test results and are, again, barely visible. As the number of bodies to be simulated were incremented in powers of 2, the data is discrete and the x-axis has been displayed using a base-2 logarithmic scale. As a result of this the y-axis is also

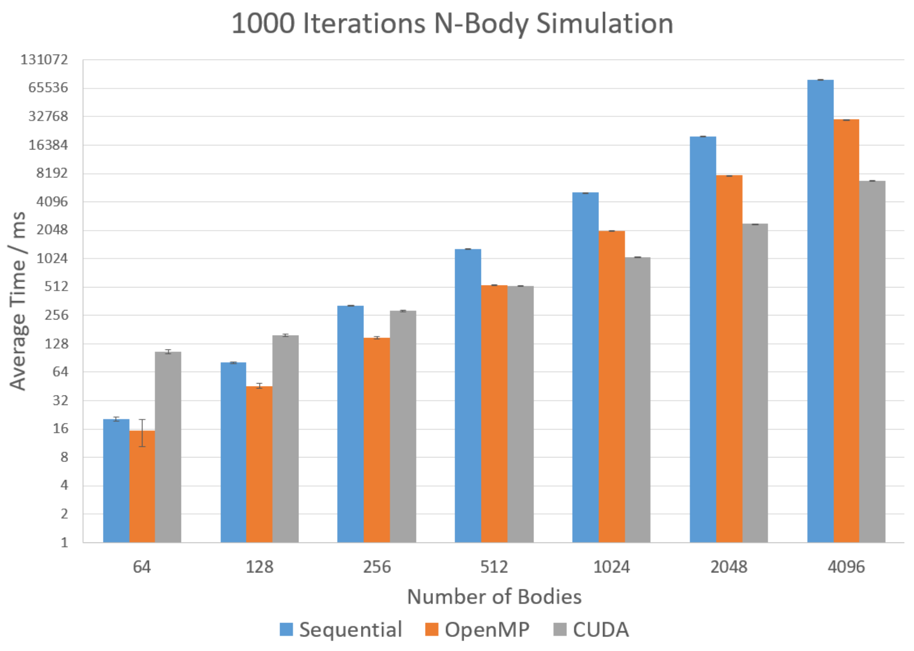


Fig. 3. A graph showing the average time it took each application to generate 1000 iterations of a simulation with differing numbers of bodies.

displayed using a base-2 logarithmic scale so that data does not appear skewed.

By running these averaged times through the formula shown in Equation 2, the speedup and efficiencies of these applications compared against the sequential application were calculated. Table VI below shows these results.

TABLE VI
ALGORITHMIC SPEEDUP AND EFFICIENCY COMPARISON

Number of Iterations = 1000				
Application	OpenMP	CUDA	OpenMP	CUDA
Number of Bodies	Speedup		Efficiency	
64	1.319	0.193	0.330	0.024
128	1.765	0.516	0.441	0.032
256	2.172	1.131	0.543	0.035
512	2.416	2.440	0.604	0.038
1024	2.550	4.814	0.637	0.038
2048	2.583	8.420	0.646	0.033
4096	2.638	11.843	0.659	0.023

This table and its accompanying graphs, Figures 4 and 5, show that for low numbers of bodies, OpenMP bested CUDA in terms of both speedup and efficiency when compared to the sequential application. When generating a simulation with 512 bodies or more, however, CUDA significantly outperformed OpenMP in terms of speedup, and was only improving as more and more bodies were simulated. In spite of these speedups, CUDA remained incredibly inefficient throughout, whilst OpenMP was nearly 3 times more efficient than the sequential application.

The positive error for speedup was determined by calculating the maximum sequential time divided by the minimum parallel time for the relevant number of bodies using their respective standard deviation error, which provides the maximum theoretical speedup. The reverse was done to determine the negative error. The speedups' maximums and minimums were then used to determine the maximum and

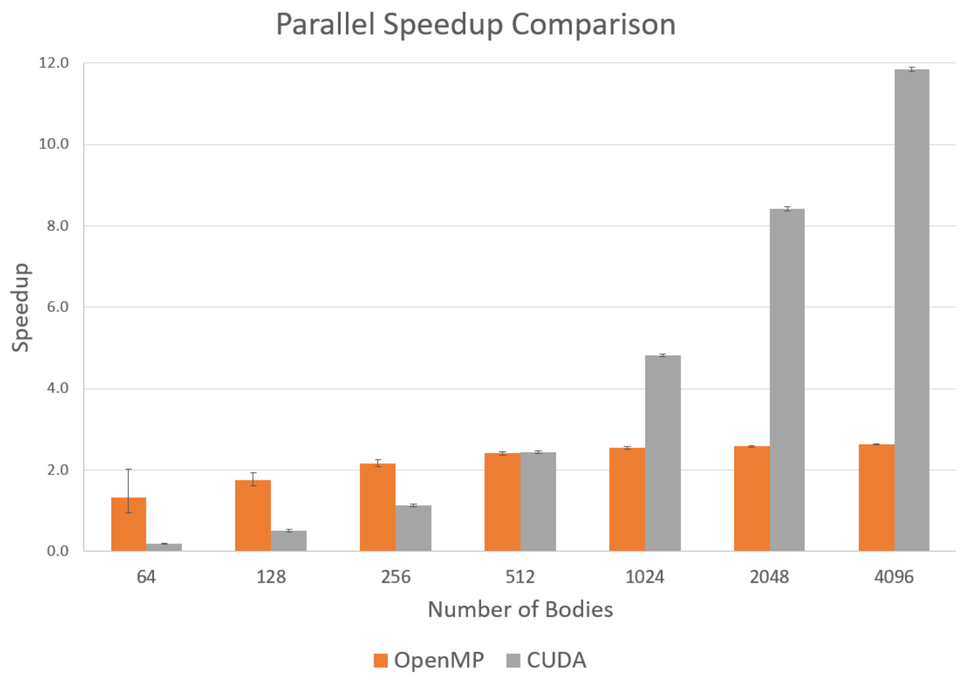


Fig. 4. A graph showing the speedup of the different parallel applications compared to the sequential application when generating 1000 iterations of a simulation with differing numbers of bodies.

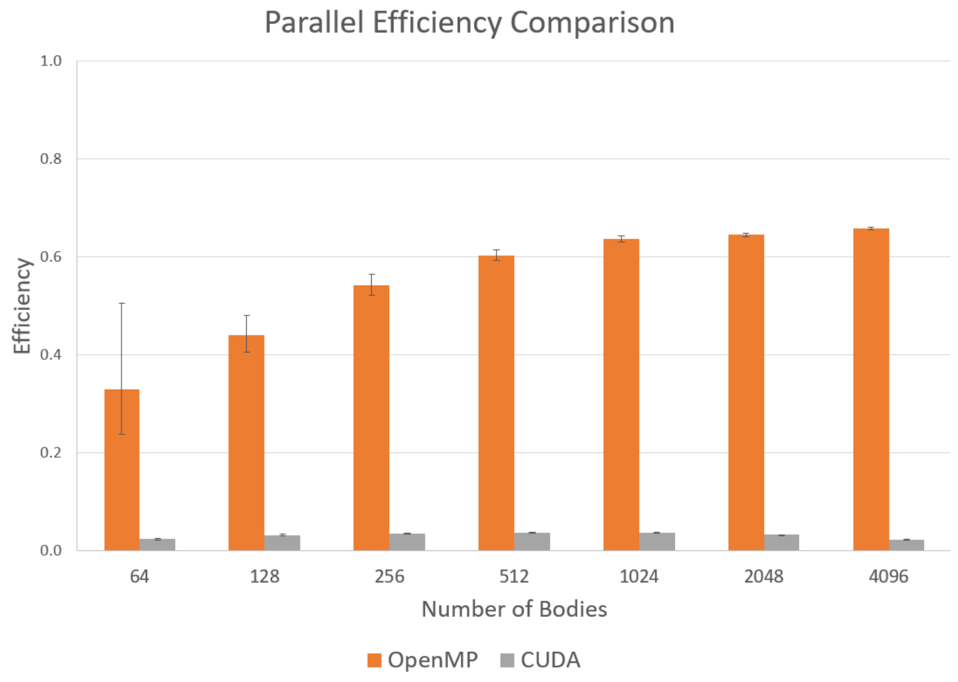


Fig. 5. A graph showing the efficiency of the different parallel applications compared to the sequential application when generating 1000 iterations of a simulation with differing numbers of bodies.

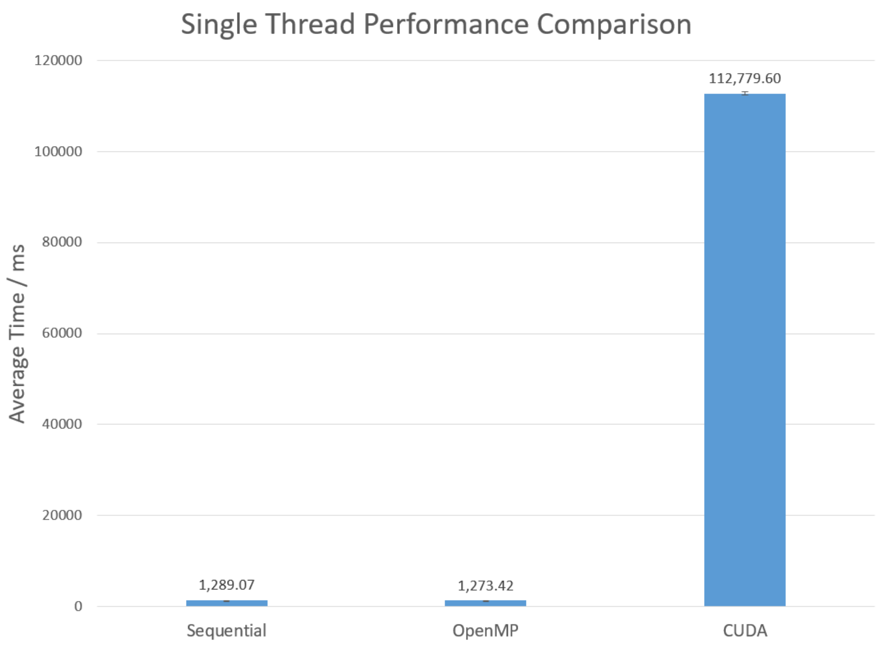


Fig. 6. A graph showing the average time it took each application to generate 1000 iterations of a simulation with 512 bodies whilst limited to a single thread.

minimum efficiency in a similar vein, which were used as their error bars. These calculated error bars are visible with a lower number of bodies, but become less significant when more bodies are added to the simulation, as is expected.

As discussed in the testing section earlier, a single threaded performance test was also performed to help contextualise these efficiencies. The results of these can be seen below in Table VII and visualised in 6. The error bars in this graph were determined by calculating the standard deviations of the test results, as before.

When limited to operating on a single thread the OpenMP application produced the simulation data faster than the sequential application showing that there were some incidental optimisations being implemented when OpenMP was used, shown in Figure 6. However, when the CUDA application was limited to a single thread, by setting both *blocks* and *threads per block* to 1 in the kernel method call, the data for the simulation took a vast amount of time to be generated. The CUDA application took longer to produce the data to simulate just 512 bodies than the sequential application took for 4096 bodies, shown in Table I. These results help to better understand why CUDA's speedups were so high, whilst its efficiency remained low.

TABLE VII
SINGLE THREAD PERFORMANCE COMPARISON

Iterations = 1000, No. of Bodies = 512	
Application	Average Time / ms
Sequential	1289.07
OpenMP	1273.42
CUDA	112779.60

V. Conclusion

A. Explanation of Results and Evaluation of Performance

Across all the tested applications, when the number of bodies to be simulated was increased, the time it took to produce the data used to generate the simulation would increase with an n^2 positive correlation. Furthermore, as shown in Figure 3, when there was a small number of bodies the OpenMP application performed slightly better than the sequential application whilst the CUDA application got outperformed by both. However, when simulating a larger and larger number of bodies, the CUDA application's performance would begin to outshine the others.

The OpenMP application had a speedup of up to 3 when compared against the sequential algorithm, particularly when simulating a larger number of bodies, as seen in Figure 4. When only generating the data to simulate a few bodies, its speedup was closer to 1. This was likely due to the fact that the average time to simulate such a small number of bodies was low, even for the sequential algorithm, and the additional time the OpenMP application would take to initialise its threads becomes a more significant factor. As soon as there was a larger amount of time spent within the parallelised $calcForces$ method, the OpenMP application's performance would increase. The CUDA application, on the other hand, had speedups of less than 1 when simulating a small number of bodies. The time taken to pass the initial n-body data from the CPU to the GPU and the time taken to pass the calculated forces back to the CPU from the GPU was a significant aspect of this. For these small numbers of bodies, the time spent inside the $calcForces$ kernel method was less than the time spent passing the data back and forth between the host and the device, resulting in processing times larger than the sequential application. When producing data to simulate a much larger number of bodies, however, the CUDA application would produce massive speedups of up to 12 (Figure 4) over the sequential application. The longer that was needed to calculate the forces to be applied to all the bodies, the better this application would perform in comparison to the others.

The efficiencies of the OpenMP application were expected, each core was less efficient than the sequential application's single core, but when run in parallel the overall performance increase was noticeable. The efficiencies of the CUDA application, on the other hand, were more interesting. Even when producing the data to simulate 4096 bodies, where its speedup was almost 12, each individual core of the GPU had efficiencies of around 0.03, when compared to the sequential application's single core, as shown in Figure 6. When limited to a single thread the reason for this low efficiency was highlighted. It took a single thread on the GPU roughly a hundred times longer to produce the same data as the sequential application did on the CPU. This was likely as a result of the significantly lower clock speed of the GPU's cores compared against the CPU, shown in the hardware specifications, Table III. Therefore, when the CUDA application was generating the data required for a large number of bodies, the kernel method was called using values of

8 for the *threads per block* and the number of bodies divided by 8 for the *blocks*, which created a large number of available cores resulting in very low efficiencies. The number of cores used to calculate the efficiency on the GPU was *blocks* multiplied by *threads per block*, which resulted in the value of p being equal to the number of bodies being simulated. Therefore, whilst the average time to produce that much data was lower than both the sequential application and the OpenMP application, and its speedup much higher, it was always significantly less efficient.

B. Final Thoughts

The written sequential algorithm originally took a considerable amount of time to produce *n*-body simulations with a large number of bodies. The attempts at parallelising the algorithm were largely successful. They both performed significantly faster than the sequential application when working with a large number of bodies and made the time to generate the data for long simulations of those bodies more manageable. However, for both parallelisation techniques, when simulating a smaller number of bodies, the improvements were either minor or non-existent. As expected, when parallelised across the CPU, the speedup was noticeable with only a slight loss in efficiency, whilst when parallelised across the GPU, the speedup was much more impressive but each individual thread was very inefficient. In terms of increased performance when desiring a long simulation of a large number of bodies, in comparison to the original sequential application, the CUDA parallelisation produced the most notable improvement.

References

[1] K. R. Meyer, *Introduction to Hamiltonian Dynamical Systems and the n-body Problem.* Springer Science and Business Media, 2009.

[2] M. Harris, *mini-nbody*.
https://github.com/harrism/mini-nbody

[3] M. Lewis, *NBodyMutable*.
https://gist.github.com/MarkCLewis