NBody Simulation Report

Anton Mitkov

40208394@napier.ac.uk

Edinburgh Napier University - Concurrent and Parallel Systems (SET10108)

Abstract

This report serves to document the work undertaken on a Nbody simulation in order to gain a performance improvement. The goal of the work is to speed up the simulation using concurrency parallelisation and GPGPU techniques.

Keywords – NBody, Parallel, Concurrent, CUDA, Graphics

1 Introduction and Background

An Nbody simulation represents a scene consisting of an arbitrary number of bodies, the shape of which is not taken into account. This is accomplished via the use of an OpenGL rendering engine, consisting of an Application class, which handles the window creation and event pool, a Shader class, which compiles the vertex and fragment shaders used to render the scene, a Mesh class, which creates a mesh of a body, which gets rendered using pre-determined vertices and normals and a Body class, to which the mesh class is attached. The Body class handles all the movement of the body, while the mesh only the visualization. The simulation is animated using physics based movement accomplished by integration. The main point of the animation is Newtonian gravity: the force used to provide the bodies with acceleration.
The Newtonian law of gravity says that the gravitational force felt on mass mi by a single mass mj is given by

$$F_ij = \frac{G * m_i * m_j * (q_j - q_i)}{||q_j - q_i||^3} \tag{1}$$

Where **G** is the gravitational constant, **q** is the position a body and **m** is the masses of the respective body. Initially the calculation for the gravitational force between two bodies can be simple and straightforward, however the gravitational forces need to be calculated for every body in the scene. Which can be understood that for an **n** number of bodies there needs to be n^n gravitational computations. This becomes more of a problem, slowing down computation, with the increments of the number of bodies in the simulation.
The OpenGL rendering engine creates a simulation following the following steps.

- The data lists used throughout the program are defined. The vertex and fragment shader file paths are defined.
- An instance of the Application class is initialized. Initialization creates a simulation window and sets up a camera.
- A number of bodies is created using a mesh of a cube and the position is randomized in all three axis going from -500 to 500 and mass set to 10. All the bodies are pushed back into a vector containing all of them.
- The main simulation loop is started. The gravitational forces are computed and the bodies are moved trough integration. The key and mouse inputs are handles.
- The application draws the meshes of each body at the already integrated position.

The calculation of the forces for each body is done using the vector containing all the bodies. For each body in the vector the gravity is calculated between it and every other body in the vector and then summed to create an acceleration for the given body for the state of the scene.

2 Initial Analysis

Due to the goal of the project being the improvement in the Nbody gravity calculation speed, the majority of code, having to do with the simulation set up will be prioritized less. The overview of the part of the program having to do with the Newtonian gravity calculation shows potential points of parallelization. The most apparent part of the program is the fact that it can be viewed as data parallel or "embarrassingly parallel". The calculation for the gravitational forces for each body are independent and do not affect the values of any other

body, they are only used to calculate the gravity for the current body.

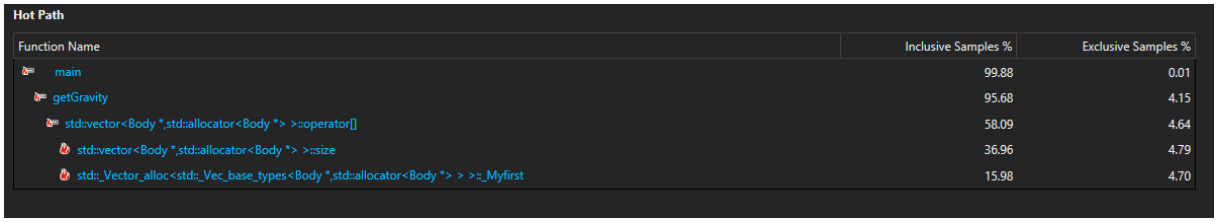

Figure 1: **VS Profiler Tool** - showcases the most work intensive part of the program, which with potential parallel optimization can produce speed up.

For additional code analysis the VS profiler tool was used (Fig 1) . It highlights the "hot path" where the most work is done in the program. As per the overview the getGravity class, which calculates the gravitational forces is one of the most work intensive methods in the code. The result was expected given the nested loops it is comprised of, which iterate for every body in the simulation.

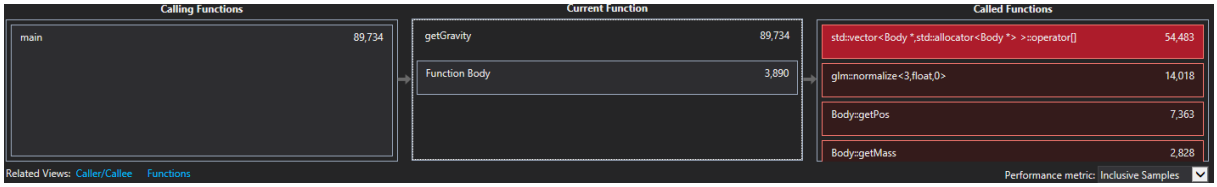

Figure 2: **Most intense method** used for the calculation of Newtonian gravity for each body of the simulation

The most used calls in the getGravity method (Fig 2) are the ones accessing the information for a specific body from the simulation. However this is required, due to the nature of the calculations for a specific body require access to the position and mass values of all other bodies in the scene.

Given the information provided by the VS Profiler Tool the potential parallelization points are confirmed. The parallelization of the loops for gravity calculation would increase the rate at which the the forces are calculated. Also, because of the communication between each bodies do not require any changes made to any other bodies apart from the one worked on, this allows

the parallelization to be done without any worry towards race conditions and non-determinism.

Num of Bodies	Mean (Average) ms	Standard Deviation	Standard Error
512	2.880765	1.06808	0.106808
1024	13.28907	5.2452077	0.52452077
1536	30.135062	9.229245	0.9229245
2048	50.813385	2.194338	0.219434

Table 1: Statistics gained from multiple runs of the sequential version of the simulation.

Table 1 was generated from from running the simulation and outputting the time required for the program to compute the gravitational forces for all bodies for one frame of the simulation. The more bodies are included in the simulation the more time the simulation takes to compute the gravity. This is showed again in Fig 3.

Figure 3: **Graph about the time** - More bodies, more time required for computation

3 Methodology

The initial analysis highlighted a main point of parallelization. The given point makes it easy for a number of parallelization approaches. Because the gravitational acceleration is computed on a per body method, using the rest of the bodies in the simulation only for a look up without influencing them, there is a potential for the computation to be done in separate threads simultaneously. This approach is viable due to the data being easily vectorizable, and as long

as the threads have access to the data for each body in the simulation the computation can be done for one body separately without any issues of non-determinism arising.

3.1 Parallel threads

For an initial approach the use of CPU threads was chosen in order to parallelize the computation of gravitational accelerations for the bodies in the simulation. The loop which iterates trough each body in the scene was the main target. The idea of this approach was to divide the iterations of the loop between different threads. In this way for an **n** number of bodies and a **t** number of threads, each thread needs to compute only $(\frac{n}{t})^n$ gravitational accelerations as opposed to the n^n computations in sequential runs. Like previously stated, the computation does not affect the list of bodies as a whole, but merely creates a list of gravitational accelerations, where each co-respond to a particular body of the simulation.

3.2 CUDA

Due to the graphical nature of the simulation a GPU solution was highly desirable. The data worked on is in a format which fits the work model of a GPU: the same simple mathematical computations needed to be done on a large group of data. Since calculation of the forces are done per body using the information about the other bodies the problem is highly data parallel. However, working with CUDA kernels creates an overhead which need to be taken into consideration and mitigated if possible. Most important of the overheads is the memory copying of data from the host(CPU) to the device(GPU) is very slow.

3.2.1 CUDA - Bruteforce

The initial CUDA approach consists of passing the required data for the computation to the GPU kernel for every frame of the simulation. The kernel would compute the gravity data for every frame from the data it is passed, and store the result in a list. The list is then copied out from the device to the host. After the data is retrieved from the kernel the GPU memory is freed, so the kernel can be re-loaded with the updated data for the bodies. The approach provided a valid result but at this stage makes no attempt to avoid the overhead that comes with using CUDA kernels.

3.2.2 CUDA - Shared Memory

This approach varies from the initial CUDA implementation due to its use of the CUDA kernels shared memory. From the observations on Fig 2 it can be noticed that the most intensive part of the getGravity function is the use of the operator used to access a specific item from the vector of bodies. Being aware that kernel accesses to the global memory are slow compared to the shared memory, this approach was implemented in order to observe the significance of the difference in speed towards the simulations.

3.2.3 CUDA - GPU movement integration

This approach looks to handle the memory copying overhead created from passing data between the host and device. The way this is accomplished is by creating two separate methods which communicate with the kernel. The first method allocates memory on the GPU and copies all the data needed for the computation not only of the gravitational forces but also for the movement integration of the bodies as well. The second method only invokes the kernel and copies out from the kernel positions of the bodies, updated in the kernel. The kernel is modified to handle not only calculating the gravitational acceleration, but the integration of the movement for the bodies as well. This requires the kernel to need more information, requiring more data to be copied across to it. This, however, is done only upon initialization and the GPU memory is not freed but kept. This means that the kernel does not need updated data for the bodies, due to it updating the data itself. This allows the program to merely call the kernel with the second method, not copying any data to it, only copying data out.

4 Results and Discussion

The results shown on Figure 4 show the time the different approaches take to compute the gravitational acceleration for different number of bodies. From the results it can be concluded that the parallelization approaches improve the computations to different degrees. While the

parallel threads approach seems to be consistent, the different CUDA approaches show a change in computation time. This serves to prove the assumptions made when creating the different approaches. The bruteforce and shared memory CUDA approaches appear slower than the parallel threads approach for a small number of bodies. This demonstrates the overhead of memory copying the data required between the host and device. At larger number of bodies however, the shared memory approach proves faster the bruteforce one, requiring less time to compute, demonstrating the speed difference between memory accessing the global GPU memory and the shared memory. Regardless of the methods, the CUDA movement integration takes the least time. When compared to the parallel approach, it serves to demonstrate the significant improvements that can be made trough using CUDA kernels. Additionally, when compared to the other two CUDA approaches highlights the overhead created by the slow memory copying between the host and device.

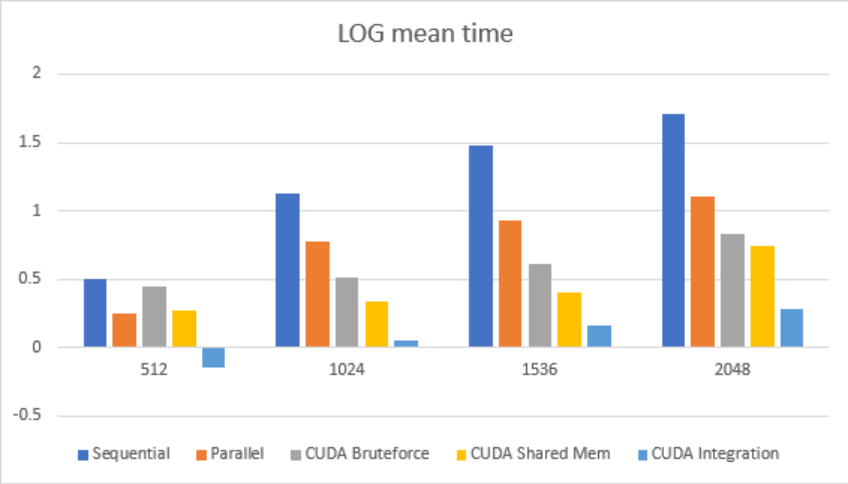

Figure 4: **Comparison of computation time** - logarithmic time comparison of the amount of time each approach requires to compute forces

Figure 5 shows the speed up achieved by the different approaches. The graph again demonstrates how the NBody simulation benefits from parallel approaches. The parallel approach seems to level out at bigger numbers of bodies. While the CUDA approaches seem only to gain speed up with the number of bodies.

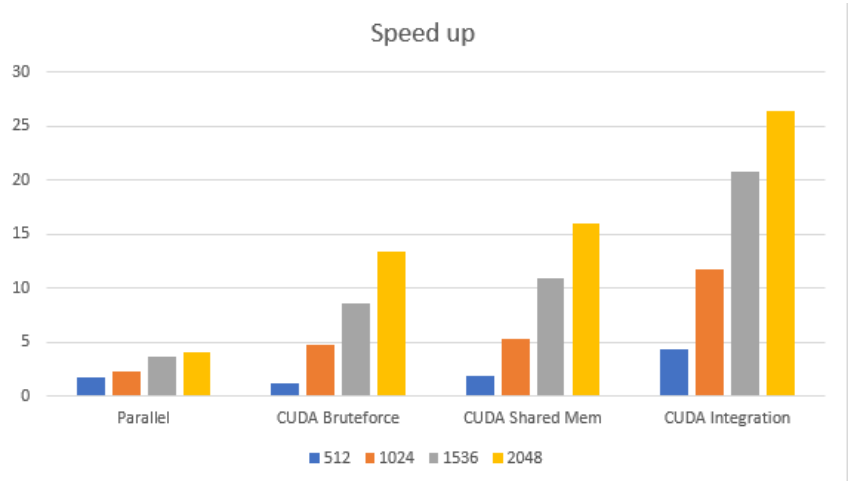

Figure 5: **Speed Up** - The proportion of the sequential time towards the given approaches.

The efficiency calculation did not make sense for the CUDA approaches. A realistic comparison cannot be made between a CPU core used in the sequential run and the parallel threads approach and the GPU cores of the CUDA approaches due to the design difference of the different type of cores. GPU cores are designed with the assumption that a large number of cores will be doing the same type of computation albeit slower than a CPU, compensating the difference in computation power with a number of computations done simultaneously, though, being the same computation.

The efficiency of the parallel computation is increased with the increase of bodies. For the parallel approach it can be argued that the initialization and joining of threads create an overhead. This means that with the increase of work a thread does the efficiency of the approach is increased. This does not scale indefinitely with the problem, the increase in efficiency diminishes at bigger body numbers.

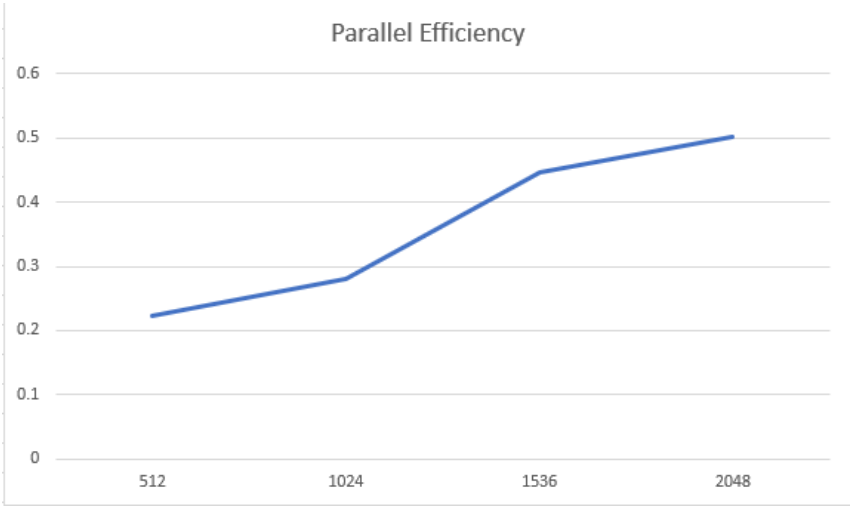

Figure 6: **Efficiency** - Efficiency of the parallel implementation.

5 Conclusion

The investigations made proves that the Nbody problem benefits greatly from parallelization. The data required for the calculation of gravity is easily vectorizable making the problem "embarrassingly parallel". The approaches taken improve the simulation immensely and scale quite well with the increase in the amount of bodies worked on. Parallel thread implementation is put into comparison to different CUDA implementations. It is seen that if implemented without taking the overheads of the approach, CUDA kernels can provide less speed up than the parallel approach. However when taken into account and attempts for mitigation made a more than 25 times speed up can be achieved with CUDA.