Parallelizing N-Body Simulation

Vishal Singh
vs2202@nyu.edu

Abstract

N-body problem is a well known simulation problem in the scientific domain which is required for simulating the effects of various objects which are interacting with one another. It has widespread uses in various technical fields ranging from the study of protein synthesis to gravitational simulations. N-body simulation is computationally intensive as it is usually performed for very large number of iterations; making it a suitable candidate for parallelization. This study investigates the parallelizability and scalability of the direct-sum N-body algorithm on GPUs and how it compares with the CPU. Parallelization on the CPU and GPU cores was achieved using the OpenMP and CUDA respectively. It was observed that the CPUs performed better when the number of bodies interacting with each other is not very large (<20000). This is expected as a typical CPU core is much more sophisticated and faster than the GPU core. It was also observed that approximations using single precision yielded much better performance gain (linear in number of memory accesses and quadratic in number of interacting objects) on the GPU compared to the CPU. The problem is inherently compute intensive which is evident from the fact that unified memory model performed roughly similar to the manually managed memory model, and that the usage of shared memory yield roughly the similar performance.

1 Introduction

The N-body problem is a problem wherein we try to estimate the effect that different objects have on each other (1). The objects usually interact with each other though a physical force and the behavior that is usually studied is the physical motion of these interacting objects. The motion largely follows Newtonian mechanics which makes a simulation based solution favorable. The solution to the N-body problem and by extension the simulation is used to study a wide range of objects such as planets, star and black holes. N-body simulation is also widely used technique to study motion of protein synthesis and design of composites (2).

In this project, Gravitational N-body problem was simulated and parallelized. In this flavor of the n-body problem, gravitational force is (usually) the primary physical force with which different objects interact with one another. The net force on any particles at any given time is given by the following equation:

$$F_j = \sum_{i \neq j} \frac{G m_i m_j}{r_{ij}^2}$$

Here F_j is the net force exerted on the j^{th} particle by all the other particles. m_p is the mass of the p^{th} particle, and r_{ij} is the distance between particle i and j, and G is the gravitational constant. Equations for motion using Newtonian mechanics can be solved by discretizing the time, by using a time step - dt. A very small dt ensures that error isn't large and that the simulation result converge. A small dt is good, however the number of iterations required will rise which may become too slow for the simulation so there is a tradeoff between speed and accuracy. For this project, I focused on parallelizing based on the assumption that dt (time step parameter henceforth) is fixed.

The motion of any particle can be described by the following equation:

$$a = \frac{F}{m} \tag{1}$$

$$v = \int \frac{F}{m} dt \approx u + \frac{F}{m} \Delta T \tag{2}$$

$$s = ut + \frac{1}{2} a t^2 \approx u \Delta T + \frac{1}{2} \frac{F}{m} (\Delta T)^2 \tag{3}$$

Here, a is the acceleration, u is the velocity before the iteration and v is the velocity after iteration and s is the distance. It must be noted that the distance assumes that the velocity is constant for ΔT time, which is not the case and better approximations are possible with similar computational complexity.

2 Proposed Idea

The primary goal of this project is to parallelize direct sum (aka particle-particle) approach for speeding up

the Gravitational N-Body simulation without algorithmic approximation and to check scalability and scaling efficiency on both CPUs and GPUs, and to see how CPU-GPU high latency reads affect performance. Here approximations refers to the approximations in the taylor series approximations and other methods such as dipole methods which approximate the net force acting on a single body (and not the floating point approximations of the numbers).

3 Related Work

The N-body simulation is straightforward in principle and has a $O(N^2)$ direct sum (aka particle-particle) solution. As the number of particles increase this approach becomes relatively slow (8). There are other faster methods, such as particle mesh - $O(N_g log\ N_g)$, where N_g are number of grid points, fast-multiple method - $O(N)$ (5).

Usually hierarchical structures are used to implement approximation method wherein majority of the influence can be approximated with nearby particles thereby reducing both computation and communication.

The state of the art algorithms in N-body simulation uses both temporal and spatial parallelization on approximation algorithms with some spectral deferred correction, such that approximations don't deviate much from the exact results by making error(s) smaller in simulation. (6).

The N-body simulation is an "embarrassingly parallel problem", and research talks about how to parallelize when number of unknown falls too low (i.e. problem becomes communication intensive when number of nodes are really large) (6). There are various hierarchical methods used for mitigating this, however venturing into these approximation, although important for the overall speedup of the simulation, are counterproductive to this project which is ultimately aimed at understanding parallelizability of the algorithm on CPUs and GPUs (8). Devising a better parallelizable algorithm is a part of multi-core programming, however, sophisticated N-body approximation and error correction is a sub-project in itself and may consume too much time. One of my goal is to see how much improvement I can get for an embarrassingly parallel problem on GPUs, and since a GPU core isn't as powerful as a CPU core, how much earlier will communication becoming an overhead. This is roughly similar to what (8) investigates.

4 Experimental Setup

The tests were performed on Prince (NYU's HPC) and my personal machine. The results on HPC were varying a lot (resource allocation was time consuming and time shared) making timing measurements highly variable and unreliable) and hence only the results on my machine are reported. Configuration for the system used is given below:

CPU	i7-8750 @ 2.2Ghz
RAM	16 GB
OpenGL	4.6
OpenMP	4.5
gcc/g++	7.5
GPU	GTX 1050
CUDA	10.2

Table 1: Machine Configuration

One of the use case for N-body simulation is to visualize the motion of different objects in real-time (Universe sandbox for instance), and for these cases as long as time taken (T) for performing N iterations with time step T_s is such that $NT_s \geq T$ then parallelization is good enough to get the frames per second to the monitor such that the video will look continuous.

However for this project, I focused on getting the final position of the bodies after certain time/iterations. The wall clock time it takes to compute this position is the measure which has been used for comparison. The number of iterations unless otherwise state is one.

There are three main versions of the program:

- Serial: This is the baseline version of the program which has the most basic but efficient implementation of the direct-sum algorithm.
- OpenMP: This uses the same algorithm as the serial version, however it (can) use more than one CPU thread for speeding up the computation.
- CUDA: This version use the same algorithm as the serial version, however it can distribute the computation across multiple GPU threads.

Checking correctness of the Nbody simulation was slightly tricky when accelerating on GPUs. Two following steps were performed for ensuring the correctness of the method.

- Checking position and velocity vector manually when only two objects are interacting with one another.
- Checking the movement of interacting bodies on the GUI and to ensure that bodies aren't jumping from one position to another and/or behaving weirdly.
- Matching the final position vectors of the serial and parallel implementation.

Figure 1 below shows a snapshot of the simulation where each colored sphere represent a body which is interacting with all other bodies with its movement being governed by equation 3. The simulation is rendered using the OpenGL library using the integrated graphics (Intel UHD 630).

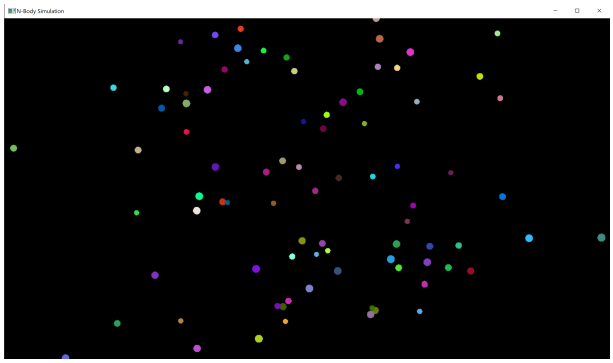

Figure 1

5 Experiment Analysis

Comparison of Serial Program, OpenMP and CUDA version

In this comparison, compute time and speedup was measured and compared across all three programs, only the time spent in computing iterations is recorded. In case of CUDA the time taken to copy final positions of the bodies from GPU to CPU is included. The optimal number of threads for openMP program was roughly around 30-40 for most invocations, and for CUDA the optimal number of threads hovered around one tenth of the number of bodies. The program is strongly scalable until the optimal number of threads and weakly scalable after this. The program is compute intensive as there are a lot of floating point operations (double precision) and is evident from the fact that both CUDA and OpenMP version performed much better than the serial version.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 71.72      6.01     6.01                             _mcount_private
  7.16      6.61     0.60 39883945     0.00     0.00  computeForce(Body&, Body&)
  7.16      7.21     0.60                             __fentry__
  4.18      7.56     0.35 44551083     0.00     0.00  Vec3::multiply(double)
  3.70      7.87     0.31 41921805     0.00     0.00  Vec3::Vec3(double, double, double)
  3.46      8.16     0.29 42766616     0.00     0.00  Vec3::add(Vec3)
  0.84      8.23     0.07 41849582     0.00     0.00  distance(Vec3&, Vec3&)
  0.72      8.29     0.06 37379194     0.00     0.00  Vec3::normalize()
  0.48      8.33     0.04                             sqrt
  0.36      8.36     0.03                             _GLOBAL__sub_I_ITERATIONS
  0.12      8.37     0.01 42180691     0.00     0.00  Vec3::magnitude()
  0.12      8.38     0.01        1    10.00    10.30  generateBodies()
  0.00      8.38     0.00    90000     0.00     0.00  generateRandomNumber(double, double)
  0.00      8.38     0.00    10000     0.00     0.00  Body::Body(double, double, Vec3, Vec3, Vec3, Vec3)
```

Figure 2: Gprof analysis of the openMP program

Figure 2 shows the gprof profiling result of the serial version of the code and from this we can see that the computations are spread across all the functions i.e. there is no bottleneck function.

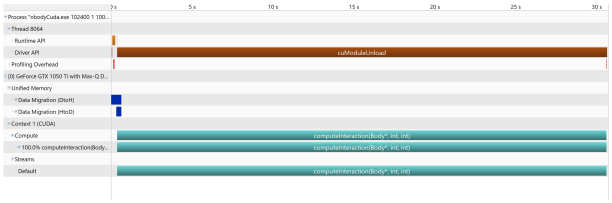

Figure 4: Profiling result with unified memory

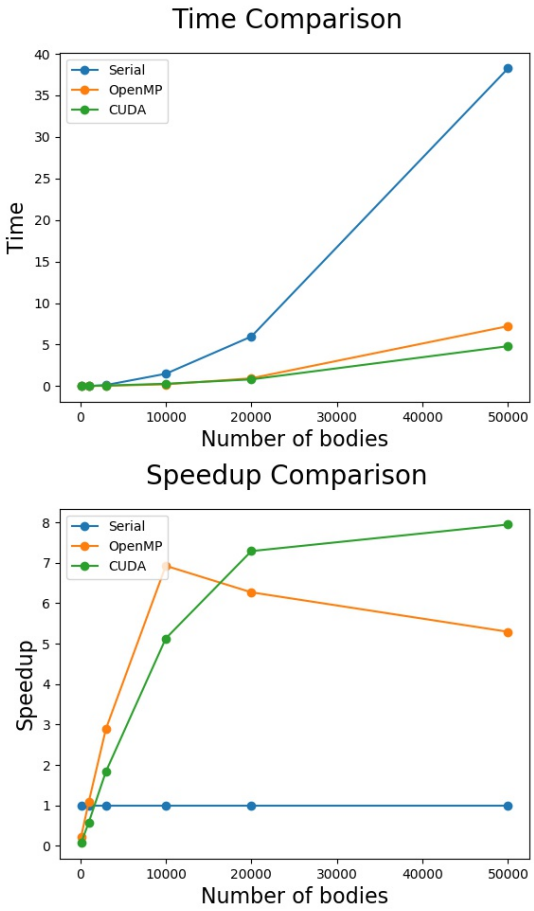

Figure 3: Speedup and Time Comparison

CUDA program was profiled using NVIDIA's Visual Profiler tool. Figure 4 shows when CUDA program was profiled. The profile shows that majority of the time was spent on computeInteraction kernel which is the main compute kernel. The profiler wasn't very helpful but we can see that there is some time spent in managing unified memory, which can potentially be further reduced.

Instead of letting CUDA manage a unified memory between the device and the host, the program was changed to allocate separate device and host memory and copying to and from device memory before and after the simulation respectively (This time was included in the measurement). Figure 6 shows the profiler result after this change and figure 5 compare the time and speedup for two versions of CUDA program, one is running cuda managed memory and another is running

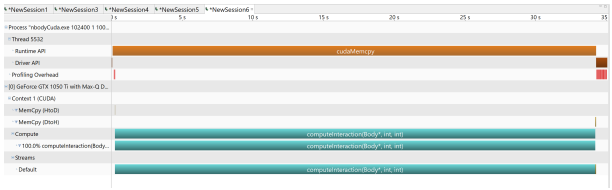

Figure 6: Profiling result with self managed memory

self managed memory. Although the profiler shows that even lesser amount of time is now spent in managing memory, the actual compute time didn't change. This could be because nvcc compiler is smart enough to see that the host doesn't touch the memory until the very end and was already optimizing it (This is supported by the shared memory test which is shown in figure 8).

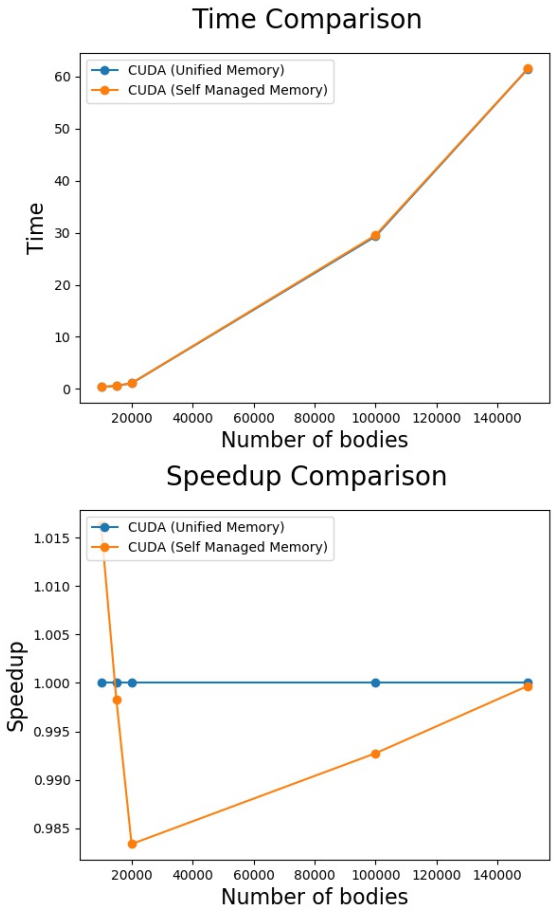

Figure 5: Speedup and Time Comparison of OpenMP and CUDA (Unified vs Self Managed memory) with Unified memory as reference for speedup

To check for scalability the same test was performed at higher body count. Since serial version of the program was taking very long time to compute it was dropped in this analysis. Figure 7 shows this comparison. Here we can see that GPU performs keeps scaling with higher and higher body count indicating that the cuda program is scalable (weakly).

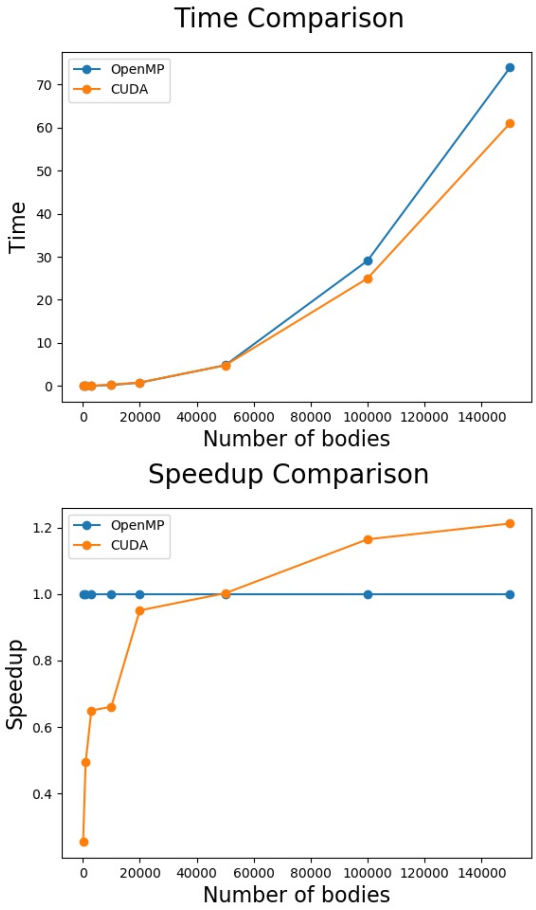

Figure 7: Speedup and Time Comparison of OpenMP and CUDA (For higher body count) with OpenMP as the reference for speedup

Shared memories are smaller and faster memories available on GPUs, and although the nbody program is compute intenstive, it was an interesting exercise to see if higher bandwidth memory can increase performance and by how much. Figure 8 compare the performance of two CUDA versions of the program, one of which use the shared memory (Time for copying into the shared memory is included) and one use the global memory. Since shared memories are very small (around 48 KB) there was a upper limit to as to how many bodies can be kept in memory, therefore the experiment was performed with ≤ 200 bodies. The simulation was done for 1000 iterations. Since the shared memory is shared only across the threads in a given block, the maximum number of threads with the same algorithm was limited to 1024, as this is the maximum number of threads that can be launched in a single block. This severely limits the parallelizability for large number of bodies. However, this can be useful when simulation involves small number of bodies but much higher number of iterations. For instance, when simulating solar system and predicting the trajectory of Jupiter/Saturn small asteroids and comets won't significantly affect its trajectory and total number of bodies would be small. The results indicate that the perfor-

mance gain using shared memory doesn't justify the constraints it puts on parallelizability and number of bodies it places on the algorithm.

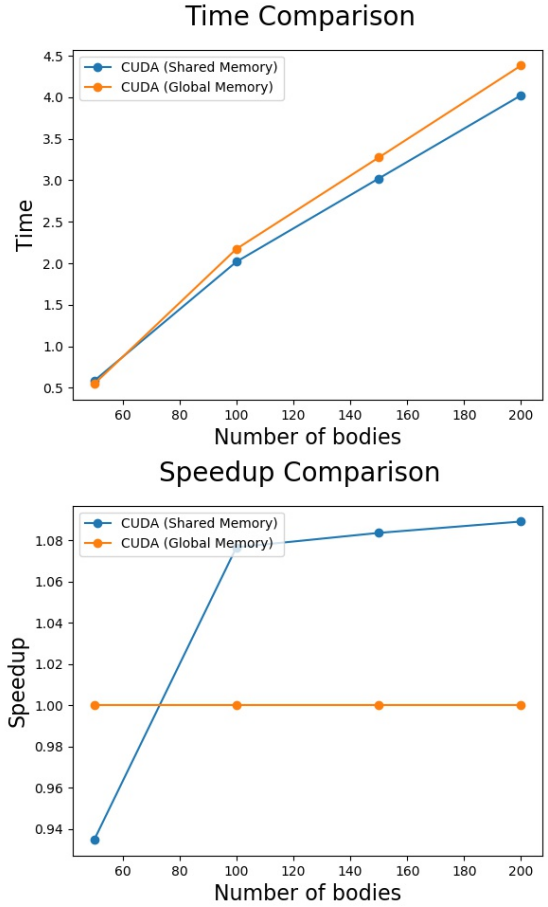

Figure 8: Speedup and Time Comparison of CUDA programs (Shared vs Global Memory) with Global Memory as the reference for speedup (1000 iterations)

Approximate Simulation

All the computations in the previous sections were done using double precision floating point numbers. It is interesting to see the performance difference when using single precision floating point numbers and -use_fast_math option of the nvcc compiler. Figure 9 shows the comparison between single and double precision. It is interesting to see that CPU didn't perform much faster when single precision is used. When using single precision GUI version didn't feel any different, and the errors for n=2 were within 5% of their position. There are some applications where the application can tolerate slight imprecision.

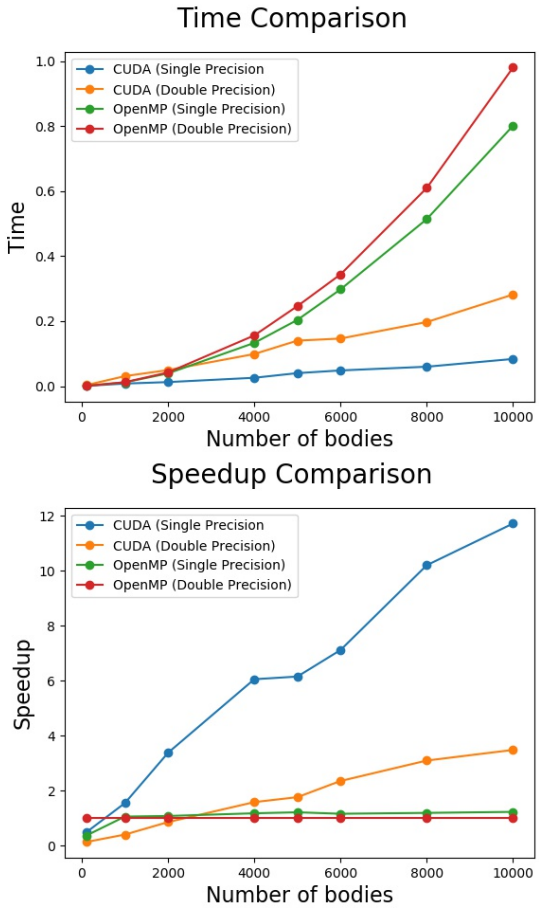

Figure 9: Speedup and Time comparison of OpenMP and CUDA (Single vs Double Precision) with OpenMP (Double Precision) as the reference.

6 Conclusion

In this report we saw that N-body can be parallelized to a large extent and GPUs are an ideal candidate for the acceleration (Speedup is around 8-12x). Global memory usage and unified memory doesn't have much impact (<10% speedup increase) on performance indicating that the simulation is compute intensive, and when accelerating on GPUs using single precision instead of double precision floating point numbers yields much better performance than single precision CPU (multi-threaded) version (≈12x speedup for 10,000 bodies).

References

[1] https://en.wikipedia.org/wiki/N-body_simulation

[2] Grama, A. Y., Kumar, V., Sameh, A. (n.d.). Scalable parallel formulations of the Barnes-Hut method for n-body simulations. Proceedings of Supercomputing '94.

[3] Amanda Randles, Efthimios Kaxiras, "A Spatio-temporal Coupling Method to Reduce the Time-to-Solution of Cardiovascular Simulations", Parallel and Distributed Processing Symposium 2014 IEEE 28th International, pp. 593-602, 2014.

[4] Edmund Bertschinger and James M. Gelb, "Cosmological N-Body Simulations," Computers in Physics, Mar/Apr 1991, pp 164-179.

[5] Hu, Y. Charlie, and S. Lennart Johnsson. "On the Accuracy of Poisson's Formula Based N-Body Algorithms." (1996).

[6] R. Speck et al., "A massively space-time parallel N-body solver," SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, 2012, pp. 1-11.

[7] Singh, Jaswinder Pal. "Parallel hierarchical N-body methods and their implications for multiprocessors." (1993).

[8] Chinchilla F, Gamblin T, Sommervoll M, Prins JF. Parallel n-body simulation using GPUs. Department of Computer Science, University of North Carolina at Chapel Hill TR04-032. 2004 Dec.