N-Body Simulation with Brute-Force and Barnes-Hut Algorithms Implemented on GPU and CPU

Kuan Chen
Yisi Lu
Qingqing Li

Abstract

N-Body Simulation is widely used as a dynamic simulation for system of particles. In this report, we present our N-Body Simulation with both Brute-Force and Barnes-Hut algorithms on CPU as well as GPU. We wrote several versions using both C++ and C language. The most fun part for this project is the simulation using openGL which gives us a nice star view.

1. Introduction

In physics and astronomy, an N-Body Simulation is a simulation of a dynamic system of particles, usually under the influence of physical forces, such as gravity. *N*-body simulations are widely used tools in astrophysics, from investigating the dynamics of few-body systems like the Earth-Moon-Sun system to understanding the evolution of the large-scale-structure of the universe [1]. In physical cosmology, *N*-body simulations are used to study processes of non-linear structure formation such as galaxy filaments and galaxy halos from the influence of dark matter. Direct *N*-body simulations are used to study the dynamical evolution of star clusters.[2]

When thinking of N-Body Simulation, we think it would be nice if we can use classes in C++. So we wrote the Brute-Force algorithm and Barnes-Hut algorithm first in C++. For brute-force algorithm, we calculate one body's force by adding all the forces from other bodies one by one after we randomly generated the bodies. Then we update the position using the force we calculated and the velocities the bodies have. For Barnes-Hut Algorithm, we have a first generate a quad tree with its total mass and center of mass of every quad node. Then we use the quad tree to approximate the force. The last step is to update the position, which is the same with brute-force method. We then implemented a C code version of this N-body simulation finding that it can be 1.5 faster than what we did in C++.

The number of bodies can be large in our CPU version of N-Body Simulation, however, the speed gets awfully slow when it comes to even 4,000 bodies. We want to use openGL to give us the evolvement with time, which means that we need to find a faster way. Then we figured out GPU can give us better performance. We can now do a simulation of 100,000 bodies with GPU. The Figure below shows a simulation of 10,000 bodies.

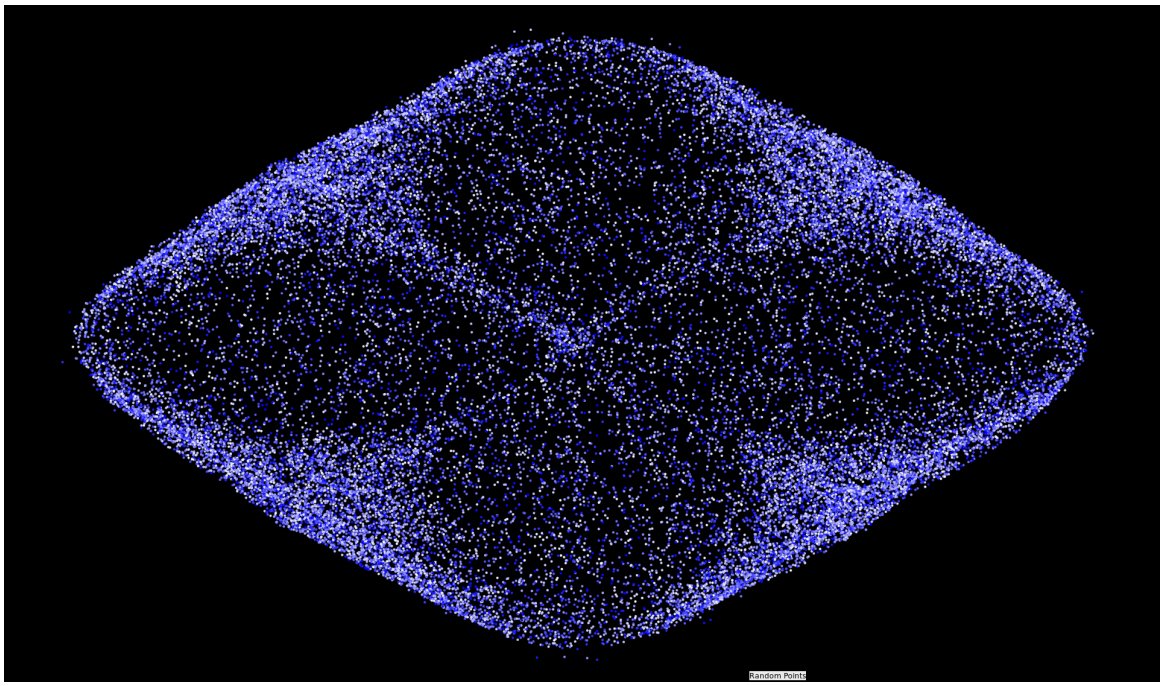


Figure 0: 10,000 Body OpenGL simulation snapshot.

2. Algorithms

2.1 Brute Force Method

The idea of N-Body simulation using brute force method is really simple.

2.1.1 Body Generation

The first step is to generate as many bodies as we want to have in the simulation process. This is done by randomly generating bodies using float number range from 1 to 10000 for body positions. We tried float number range from 1 to 1000 at the first time which shows great difficulty because many of the bodies seem to be really close to each other which cause difficulty to simulate. We initialize the bodies with a 2D coordinate and a mass.

2.1.2 Force Calculation

As every two bodies have gravitational force (as is shown in Figure.1) between each other, for each body we calculate the force from all the other bodies in this N-Body galaxy. Then we add up all the 'N' forces for one body and iterate the whole process for all N bodies. So we can see from this process that we need to do N^2 calculations for calculating all the forces for all the bodies. After calculating each body's force, we use that force for updating each body's new position.

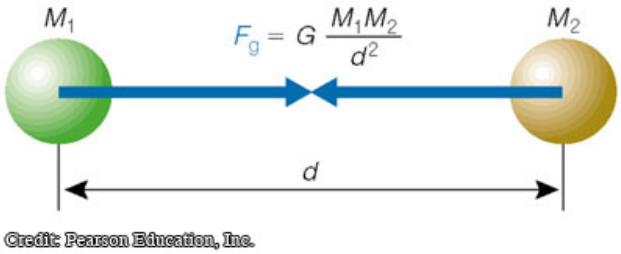


Figure.1 Gravitational Force representation

We can then derive the equations for calculating all the forces and each body's force can be represented as the following equation:

$$F_i = \sum_{j=1, j\neq i}^{N} \frac{Gm_i m_j}{d_{ij}^2}$$

F_i --The force of body i
M_i --Mass for body i
M_j – Mass for body j

2.1.3 Update Positions

After successfully calculate the force, we can then update the positions for all the bodies during a certain time interval. As we are evaluating this using a very small time interval, we assume that during this time period, the force does not change. Then we do the math and get the velocities of all the boides and use velocity and time to get the new positions.

The equations are shown below:

$$V_x = F_i \cdot \cos i \cdot time$$
$$V_y = F_i \cdot \sin i \cdot time$$

V_x is the velocity of body i along x coordinate
V_y is the velocity for body i along y coordinate

$$x_{new} = V_x \cdot time + x_{old}$$
$$y_{new} = V_y \cdot time + y_{old}$$

The two equations above show how we get the new position for a body. For our algorithms, we will need to get the new position for every body after force calculations, so there are 4*n calculations for all the bodies. The time complexity is O(n) in this problem.

2.2 Barnes-Hut Method

Barnes hut simulation is an approximation for the brute force simulation, which use the idea that we can combine nearby bodies together and approximate them as a single body. If the group is sufficiently far away, we can then we can approximate its gravitational effects by using its center of mass. [3] There are mainly four steps after initializing all the bodies, namely quad-tree creation, mass

distribution computation, force calculation and position calculation

2.2.1 Quad-Tree Creation

In order to group the nearby bodies together, we need to have a good structure to define whether they are close and a good structure to represent mass for a nearby cluster of bodies. Barnes hut algorithm uses quad tree to do implement the approximation. A quad tree is tree data structure in which each internal node has exactly four children. [4]

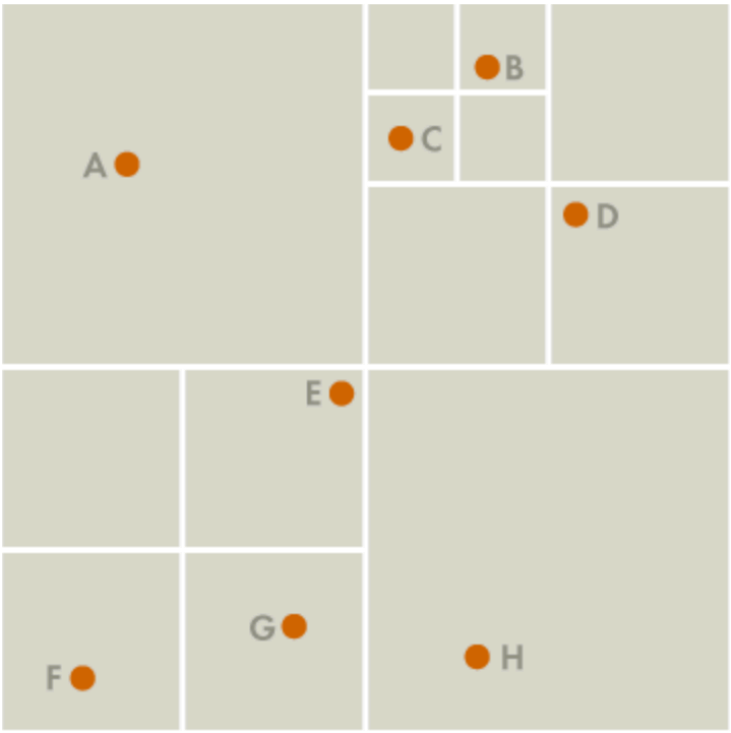


Figure.2 Quad Tree 2D representation

For a quad tree configuration, the root quad node is the whole drawing space which can be divided according to the conditions of its children. So our approach is that we have a QuadNode class in C++, we add a body from the root node each time. First condition is when the quad node is empty and it does not have children, you just add the body in this node, and this body's mass and center of mass represent the quad node's mass and center of mass. The second condition is when we are inserting a body to a node that has a body in it. The third condition is that we add can add the body to one of its four children of the quad node. We then split the node into four parts and then add the original body and new body into the children node. Then calculate the mass using the equations showing below. A tree-like representation for a quad-tree is shown in Figure.3.

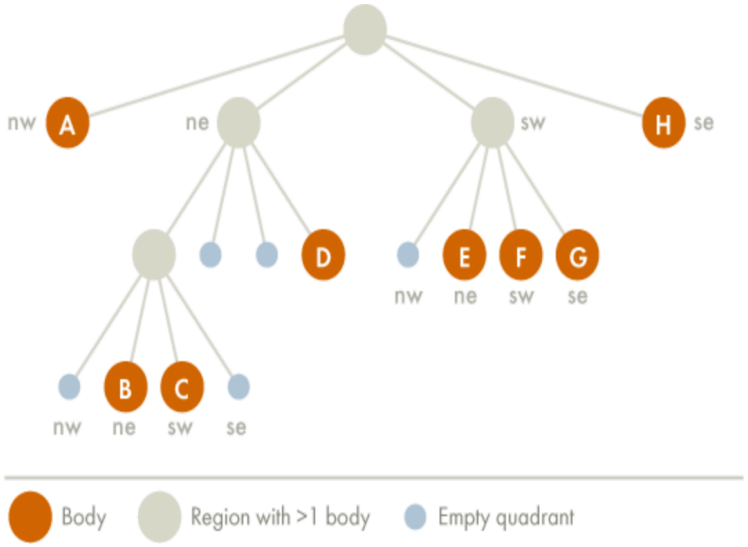


Figure.3 Quad Tree representation

The grey ones represents node without children. The orange ones represents nodes that have a child. For our algorithm, we calculate the mass distribution every time we add a body into a quad node. So we then have the mass and center of mass for all quad nodes after we inserted all bodies. The equations are:

$$m = m_1 + m_2$$

$$x = (x_1 \cdot m_1 + x_2 \cdot m_2)/m$$

$$y = (y_1 \cdot m_1 + y_2 \cdot m_2)/m$$

These equations show how we calculate the center of mass and total mass for a quad node if we have two bodies in this quad node. It is similar if we have n bodies in this quad node.

As we use a tree structure in Barnes-Hut algorithm, the average depth of the tree is O(logn). Time complexity for quad tree creation and mass calculation is then O(logn).

2.2.2 Force Calculation

We obtained a quad tree data structure from the last part. In this part we will use approximation based on the ratio of the distance to quad node height. If the ratio is bigger than a certain threshold, we then take it as far away enough for us to view this node as a single body. If not, we divide the quad node and calculate the force separately among the four nodes.

2.2.3 Position Calculation

Equations for positions are the same with 2.1.3(brute force method). Using the equations we can update the positions in O(n) time.

2.2.4 Our Barnes Hut alterations and complexity

In order to better parallel Barnes-hut algorithm and implement it in GPU, we modified our algorithm a little bit. Instead of generating the quad tree recursively, we now divide the root node into several levels . Each internal node will have 4 children this time. Then we use the algorithm below to calculate

which node the body fall into. The last part for creating this "Quad Tree" is to calculate the total mass and center of mass for all the leaf nodes.
Then come to GPU part, we calculate each body's force and new position in GPU. For each body, we iterate all the leaf nodes in our quad tree and add the body's force. Then update all bodies positions.

2.3 Time complexity

For Brute-force method, we need $O(n^2)$ to calculate the force and O(n) to update positions. The time complexity for Brute-force method is then $O(n^2+n) = O(n^2)$. In Barnes-Hut algorithm, we need O(nlogn) to construct the tree, but we only need O(nlogn) to calculate force. The position updating time is also O(n).So we can conclude that the time complexity for Barnes-Hut algorithm is O(nlogn+n) = O(nlogn)［5］. The theoretical figure of time complexity without consideration of factors is shown in Figure.4. x coordinate represents the number of bodies, y coordinates represents the time.

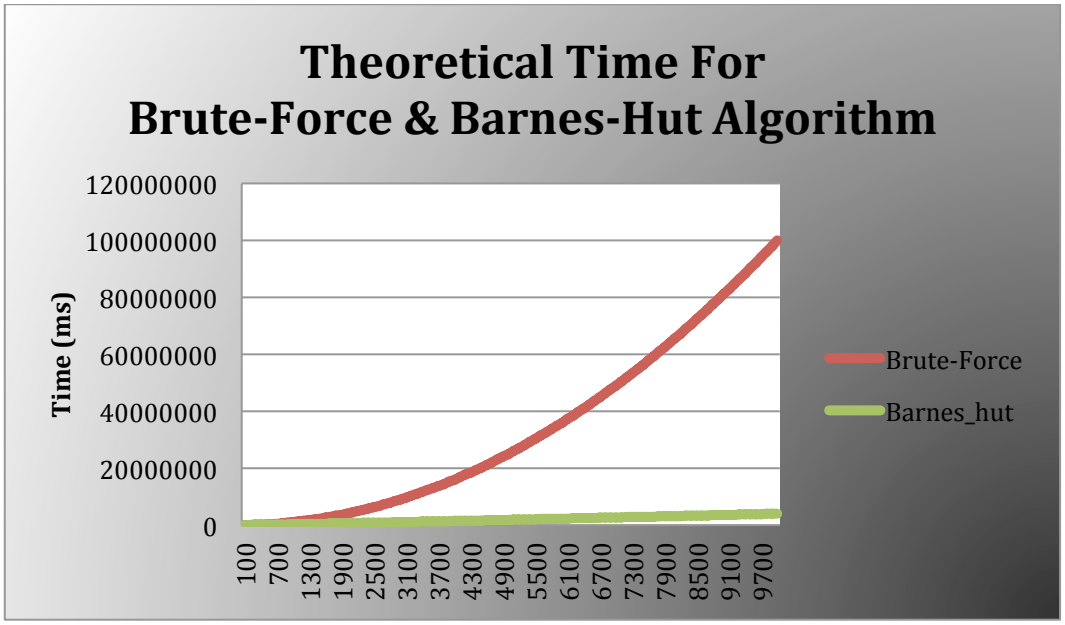


Figure.4 Theoretical time for brute-force and barnes-hut algorithm

3. Code Works

3.1 Versions of Our Code

We first implement our algorithm in C++ where we a body class, a quad node class and a simulation class. We thought it would be easier for us to think and implement. However, we found out that C++ code tend to have much more overhead than C code which would eliminate the information of performance. We then wrote versions of C code for both CPU and GPU. Table.1 shows versions of our code.

Algorithm & Method	C++	C
Brute-Force CPU	Yes	Yes
Brute-Force GPU	Yes	Yes
Barnes-Hut CPU	Yes	No
Barnes-Hut GPU	No	Yes

Table.1 Versions of N-Body-Simulation Code

We've done 6 versions of the code and we will show you more details about the performance of each version in the evaluation part below. We first went with c++ due to the object oriented nature of the language provided good structure to program. Later on we discovered the data from objects could be abstracted into 2D float array to save space. GPU global memory is limited so we reduced out footprint on the memory end as much as possible.

4. Evaluation

4.1 C++ Performance Evaluation

We run our code several times and simulate up to 10,000 bodies. We can see from Figure. 5 that when we have less than 1000 bodies, Brute-Force method is faster than Barnes-Hut method because we have a larger overhead for distributing all the bodies. The version here had a terrible performance on the GPU end because we decided on doing one thread per block for our first iteration. Later code in C++ will increase thread count to 1024. In the case below our CUDA Code performed worse than are Barnes Hut CPU code.

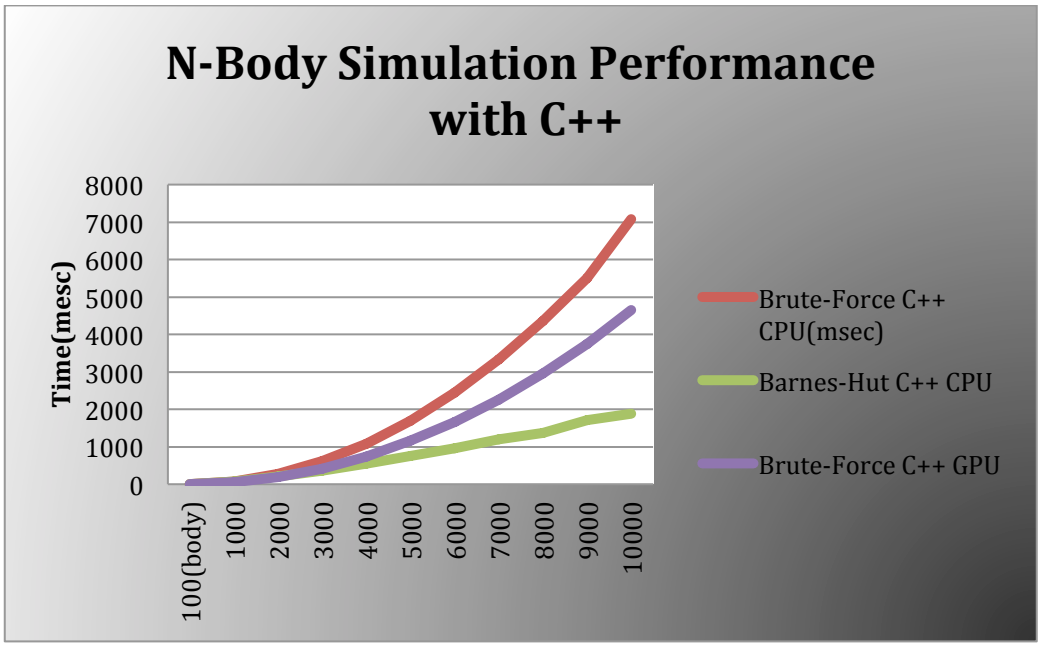


Figure.5 N-Body Simulation Performance with C++(Lower readings are better)

In order to see the performance when the bodies are few, we simulated number of bodies range from 100 to 1300 and draw the performance in Figure. 6.

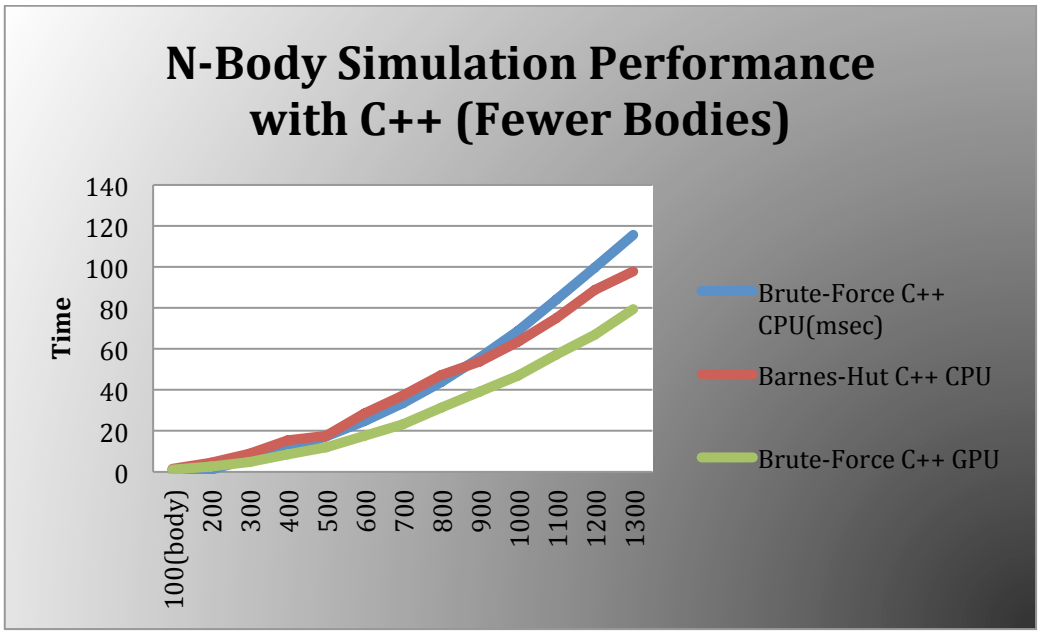


Figure.6 N-Body Simulation Performance with C++(Fewer Bodies)

When bodies reach about 900, Barnes-Hut algorithm start to perform better than Brute-Force method. However, for our Brute-Force method with GPU, it seems that the overhead is small because with body number 200 we can have better performance than Brute-Force CPU.

We then calculate the speed up based on Brute-Force C++ CPU performance, and we get the following speed up in Figure. 6

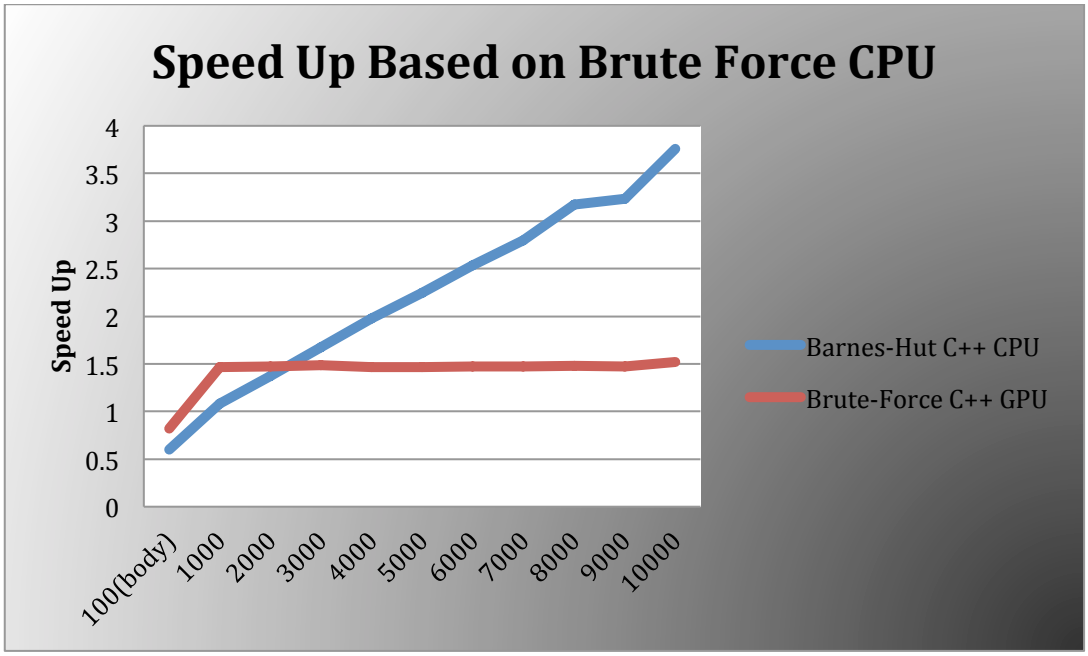


Figure.6 Speed Up Based on Brute-Force CPU Performance

Barnes-Hut algorithm implemented in CPU is obviously much faster than the Brute-Force method. As number of bodies gets larger, we can get much better performance using Barnes-Hut method. We can see that the speed up is only 0.6 when we have 100 bodies, it goes up to 4 when we have 10,000 bodies. On the other hand, our GPU speed up does not change much with only 1.5 times speed up after 1000 bodies. It is much less than what we thought, so we dig into our code.

```
__global__
void ComputeForce(Body* Bodies)
{
    //a[threadIdx.x] += b[threadIdx.x];
    int myid = blockIdx.x;
    int j;
    double x_dist,y_dist;
    double r_Squared;
    double dist;
    double Force;
    cuPrintf("hello%d\n", myid);


    Bodies[myid].fx = 0;
    Bodies[myid].fy = 0;

    for(j = 0 ; j< gridDim.x; j++){

            if(myid!=j){

                    x_dist = Bodies[myid].x - Bodies[j].x;
                    y_dist = Bodies[myid].y - Bodies[j].y;
                    r_Squared = (x_dist*x_dist)  + (y_dist*y_dist);
                    dist = sqrt(r_Squared);

                    if(dist > 10){

                        Force = (Bodies[myid].mass * Bodies[j].mass )/ (dist *dist * gridD
                        Bodies[myid].fx -= Force * x_dist ;/// dist;
                        Bodies[myid].fy -= Force * y_dist ;/// dist;

                    }
            }
    }


 __syncthreads();
```

Our GPU code in C++ version use tons of blocks with only one thread in each block for the simulation which means that we will use 10,000 blocks for the computation of 10,000 bodies. This is mainly the reason why we don't get a much higher performance using GPU. So we modified our code in C code using 1024 threads for each block in GPU. This version of our code for CUDA was running with 1024 threads per block however many blocks it took to process all elements.

4.2 C Performance Evaluation

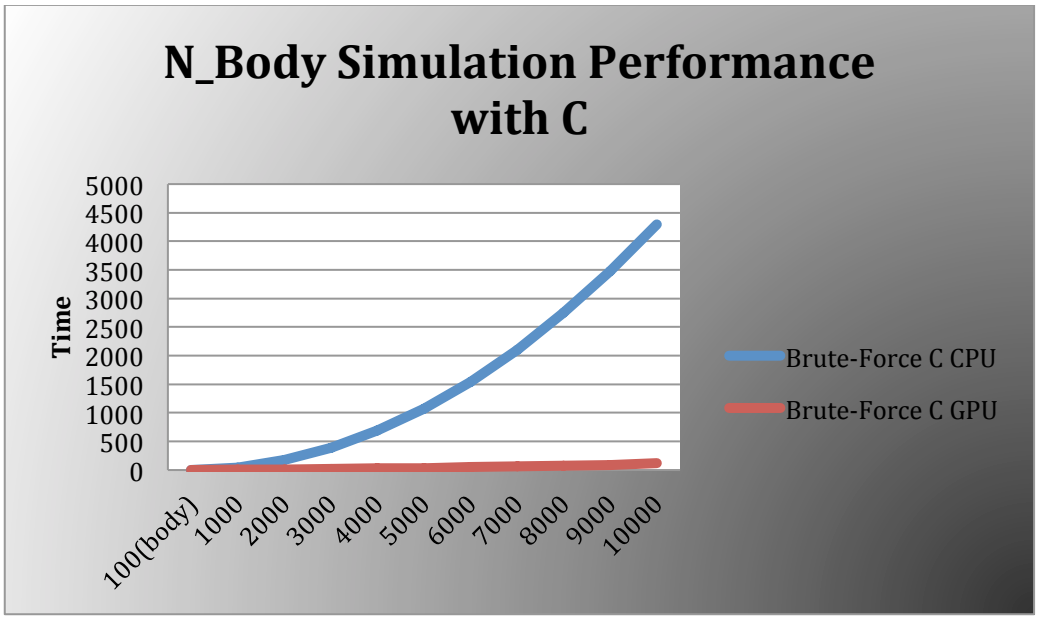


Figure.7 N-Body Simulation Performance with C (lower readings the better)

Great performance is made through implementation of GPU using 1024 threads per block. We can also see from Figure.8 that the overhead for Brute-Force GPU is very small, GPU performs much better than CPU when we have more than 200 bodies. Our simulation is to 10,000 bodies also. The speed up is shown in Figure.9.

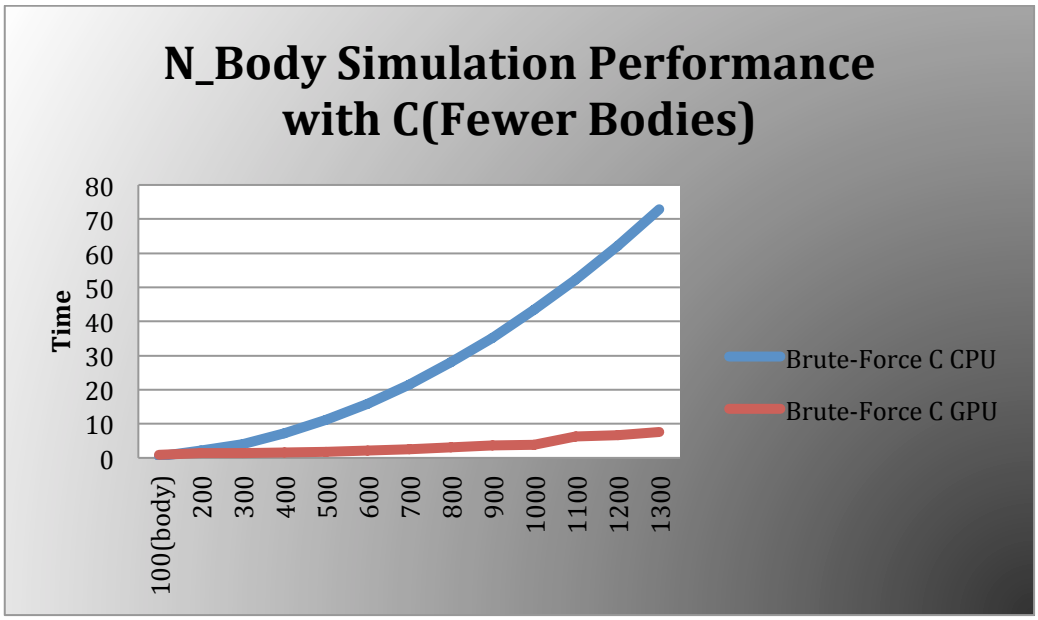


Figure.8 N-Body Simulation Performance with C (Fewer Bodies)

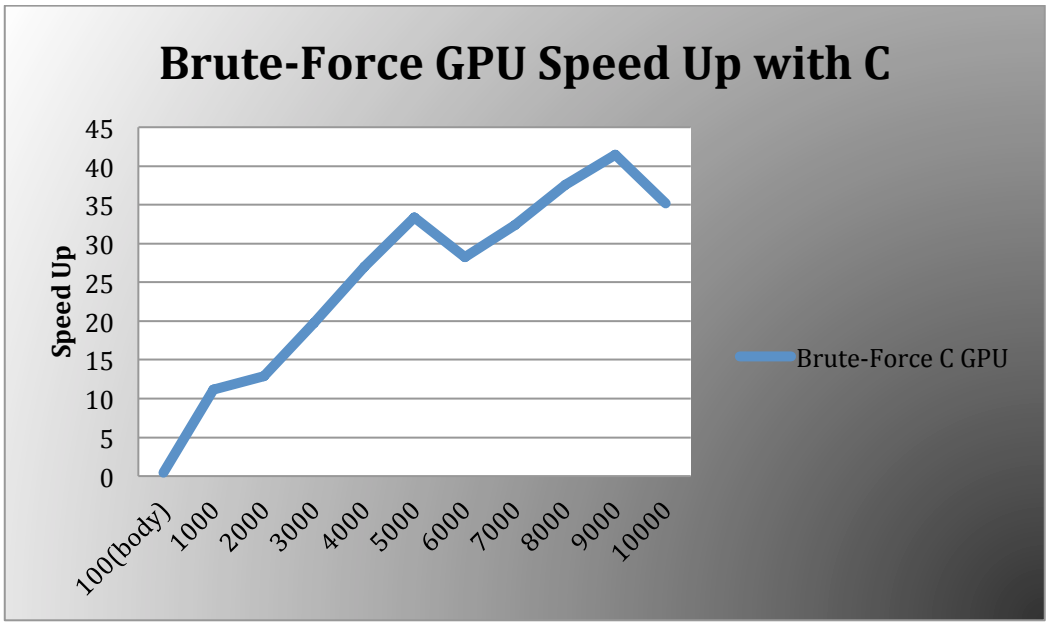


Figure.9 Brute-Force GPU speed up with C

After implementing GPU in our C code, the speed up is growing as we increase bodies. The maximum speed up is about 41 when we have 9,000 bodies. The huge increase of the performance is first the GPU implementation with multiple threads in a block and multiple blocks. We use one thread for a body in this simulation.

4.3 GPU Performance Evaluation

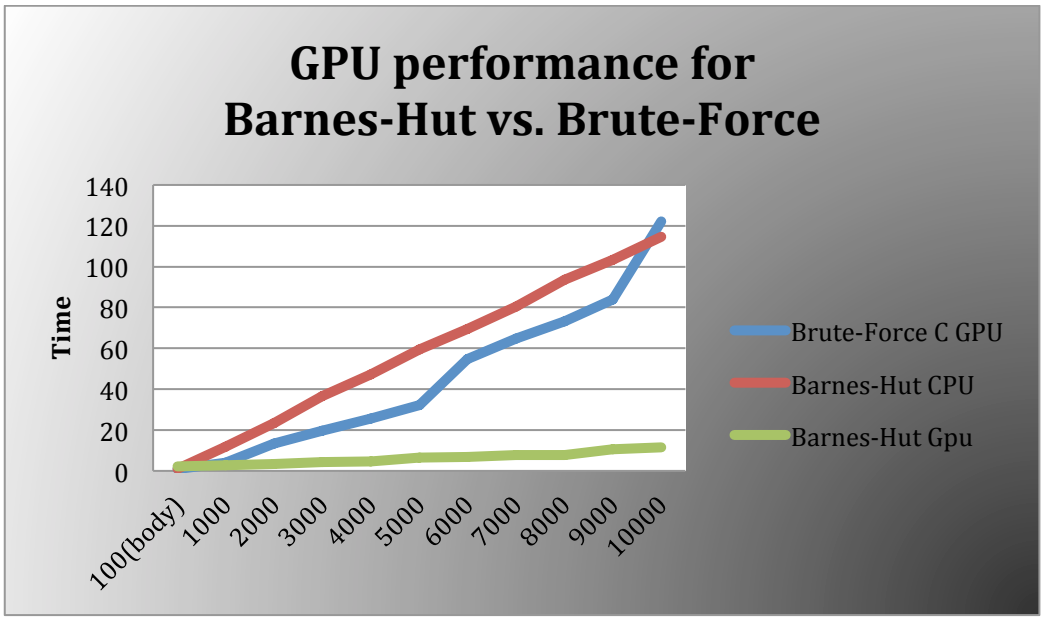
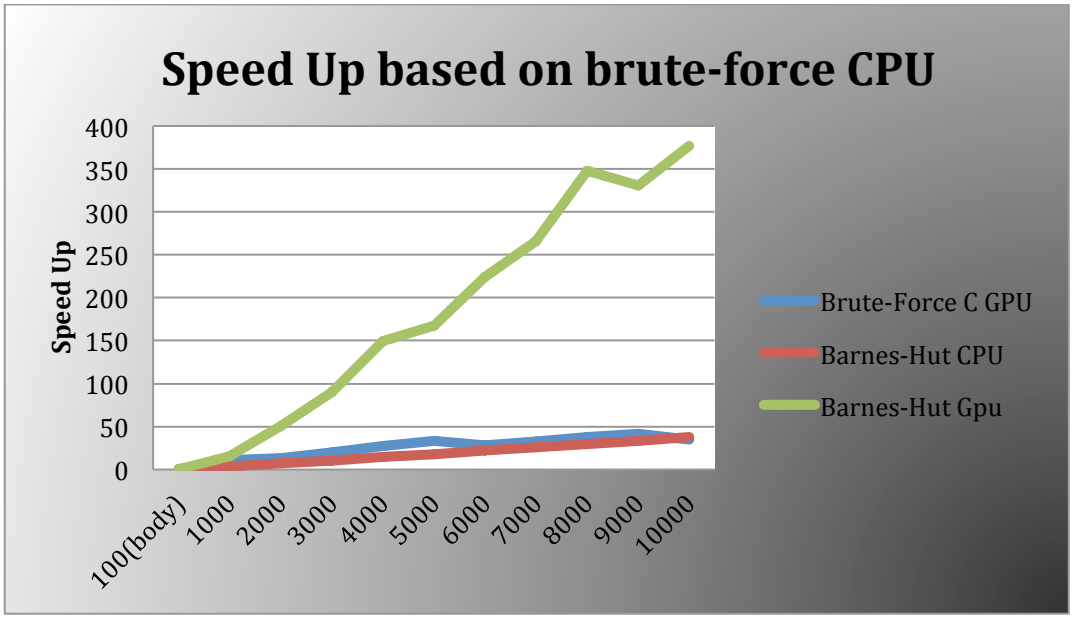


Figure. 12 GPU performance for Barnes-Hut & Brute-Force



4.4 C++ and C Comparison(Higher the better)

As we have both C++ and C versions, we made some comparisons for the executing time with same number of bodies. The result is shown in Figure.10.

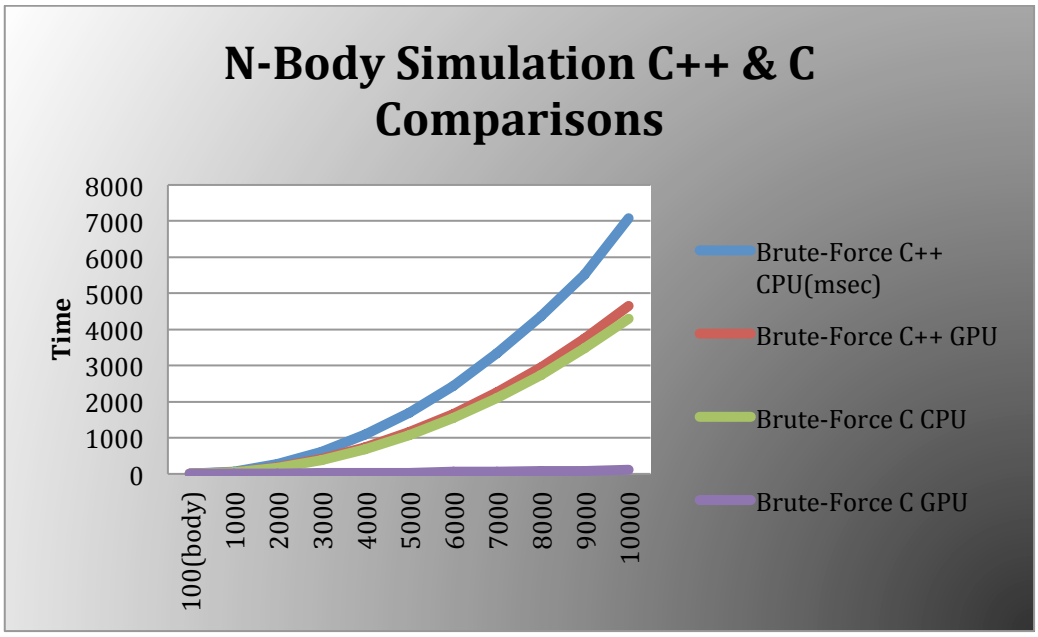


Figure.10 N-Body Simulation C++&C Code Performance Comparisons(Smaller the better)

In our experiment, the processing time for the same quantity of bodies vary a lot with C++ and C code. If we use C code for the simulation, it will save us about half the time.

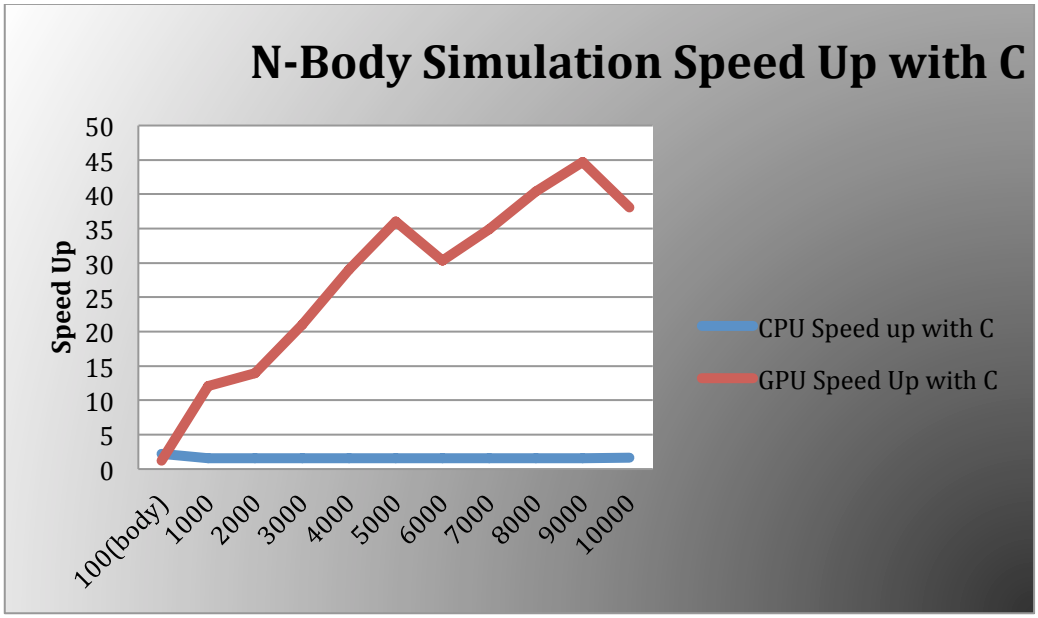


Figure. 11 N-Body Simulation Speed Up with C（Higher the better）

The blue line in Figure.11 shows the speed up of brute-force N-Body Simulation using C code. The speed up stays around 1.5, which indicates the overhead for using C++. GPU performance speed up is much bigger, mostly because we use more threads in a block when doing simulations with our C code. Our last iteration of Barnes Hut algorithm achieved an overall factor of

4.5 Results stacks together on an absolute scale!

Algorithm & Method 10,000 Particle RunTime	C++	C
Brute-Force CPU	7078.3 msec	4294 msec
Brute-Force GPU	4653.2 msec	122.1 msec
Barnes-Hut CPU	1883.2 msec	114.6 msec
Barnes-Hut GPU	Nan	11.4 msec

Future Work

We are hoping to bring all the versions we have hopefully under one roof. Also we want to improve the graphics and angle of the OpenGL. Right now our objects can follow more of a factory method in our code implementation going forward. Since Barnes hut algorithm is an estimation algorithm. We set up our code to be able be modular. This means we can switch in and our different algorithms and see the effects.

Reference

[1]Michele Trenti; Piet Hut. "N-body simulations (gravitational)". Scholarpedia. Retrieved 25 March 2014.
[2] http://en.wikipedia.org/wiki/N-body_simulation
[3] http://arborjs.org/docs/barnes-hut
[4] http://en.wikipedia.org/wiki/Quadtree
[5]J. Barnes und P. Hut: "A hierarchical O(N log N) force-calculation algorithm" in Nature,324(4) Dezember 1986
[6]Jim Demmel " Fast Hierarchical Methods for the N-body Problem", Part 1, Applications of Parallel Computers (CS267): Lecture 24, April 11, 1996
[7]Tom Ventimiglia and Kevin Wayne " Barnes-Hut Galaxy Simulator", Programming Assignment (Computer Science; COS126), 2003