Term Project Report

Multi-body Gravity System Simulation

Yuhao Xu(yuhaox1) Fangzhou Xu(fangzhox) Shuang Chen(shuangc)

Overview

Multi-body gravity system is a major problem in the field of Mechanics and Astrophysics. Simulations of this kind of system requires tracking multiple physical values (position, velocity, mass) of each body and calculating gravity interactions between each pair of bodies. Using basic mechanics principles, we can calculate the displacement of each body at each frame.

The reason we choose this problem for the term project is that it provides a simple but meaningful model for us apply multiple optimization methods. The calculation of forces over a certain body requires the information of position and mass of all other bodies (or at least for the naive algorithm), this provides a certain amount of shared knowledge among possible parallel processes. Resolving this issue should be a major target in our effort to optimize the algorithm well keeping correctness and consistency in mind. As a result, this problem provides a nice target to work out something practical and useful.

You may find the demo video at https://youtu.be/LfZ8EKnjbqY.

Related Work

Neki and Betim[1] designed a geophysical inversion of gravity anomalies running in parallel systems. Parallelization is done using both OpenMP and MPI. Results confirm the scalability of the runtime at the order of O(N8) relative to the linear size N of 3D models, while the impact of increasing the number of involved cores remains disputable when walltime is considered.

A speech about parallel gravity[2] used Graphics Processing Units (GPU) to solve the gravitational force equations. The talk starts with the embarrassingly parallel direct N-body methods, which are ideal for parallel architectures, like GPUs, to the more complex methods as used in high precision production quality astrophysical simulations.

GrAVity[3] is a massively parallel antivirus engine. It modifies ClamAV, the most popular open source antivirus software and achieves end-to-end throughput in 20 Gbits/s has 100 times the performance of the CPU-only ClamAV. Their work shows that modern graphics cards can be used effectively to perform heavy-duty anti-malware operations at speeds for N-body problems.

Soft-Sphere Discrete Element Method (SSDEM) [4] allows for the modeling of the different contact forces between particles in granular material, such as various kinds of friction, including rolling and twisting friction, and the normal and tangential deformation of colliding particles.

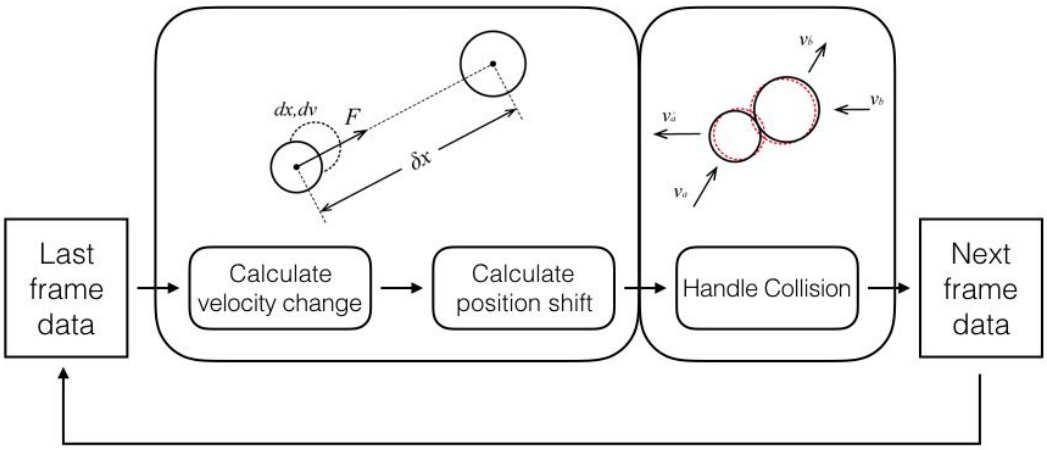
Professors from UCSC discuss collision detection and response in general, presents two collision detection algorithms[5].

Description of Techniques

1. Sequential Implementation:

The sequential implementation of this simulation algorithm is very intuitively derived from its physics meaning.

Generally, the algorithm calculates the position and velocity of each body frame by frame, where frame means a tiny period of time interval which we believe to be small enough to generate less error than we can tolerate. More specifically, the calculation process is composed by the following parts and conducted in the following order:



1. For each body, calculate its velocity change by its relation with all other bodies. For each other body, its impulse on this body's velocity is calculated by

$$dv = Gm_a\left(\frac{\delta x}{|\delta x|^3}dt + \frac{1}{2}\frac{\delta v}{|\delta x|^3}dt^2 + \frac{3}{2}(\delta x \cdot \delta v)\frac{\delta x}{|\delta x|^5}dt^2\right)$$

   where δx is the distance vector of the two bodies, and δv is the velocity difference vector of the two bodies. This equation has error less than $O((d\,\delta x)^2)$.
2. For each body, calculate its position shift by its relation with all other bodies. For each other body, its impulse on this body's position is calculated by

$$dx = Gm_a\left(\frac{1}{2}\frac{\delta x}{|\delta x|^2}dt^2 + \frac{1}{6}\frac{\delta v}{|\delta x|^2}dt^3 + \frac{1}{2}(\delta x \cdot \delta v)\frac{\delta x}{|\delta x|^5}dt^3\right)$$

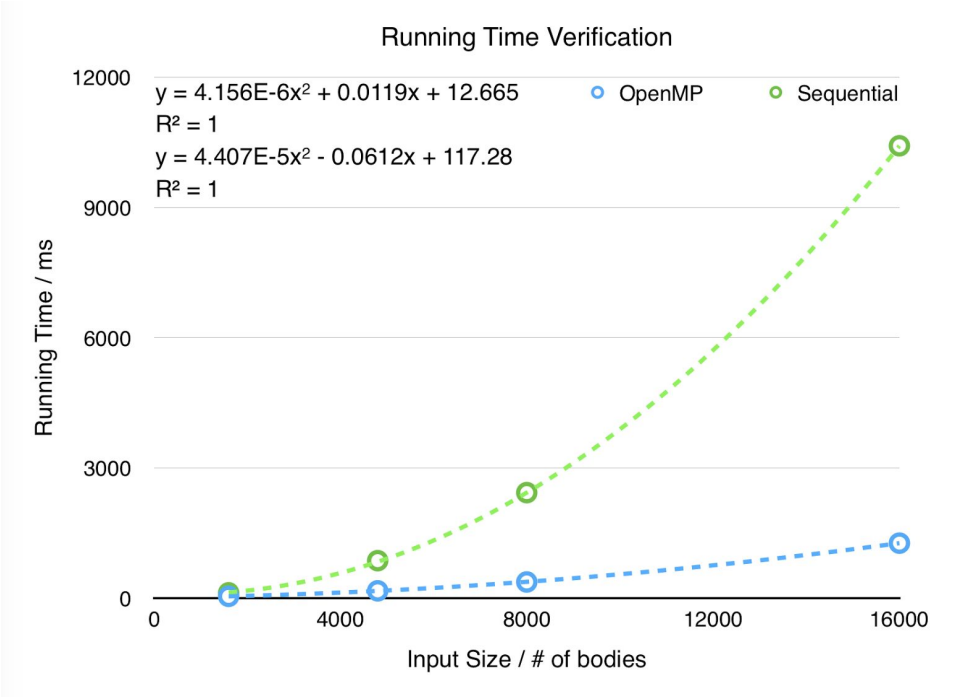
   where δx is the distance vector of the two bodies, and δv is the velocity difference vector of the two bodies.
3. For each body, check if it's collided with any other bodies. If so, calculate their velocity and position after the collision.

After these calculation, we have the correct (or at least with error tolerance) position and velocity for each body in the next frame.

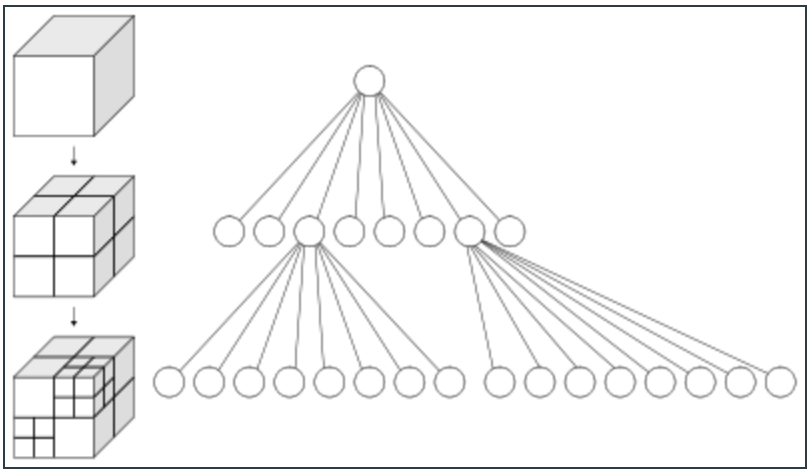
```
Vect caldv(BodyX b, BodyX a) { //impulse of a on b
    Vect dx = a.c - b.c;
    Vect dv = a.v - b.v;
    double r = dx.abs();
    return G * a.m * (
           dx / (r * r * r) * dt +
           1.0 / 2.0 * dv / (r * r * r) * dt * dt +
           3.0 / 2.0 * (dx & dv) * dx / (r * r * r * r * r) * dt * dt
           );
}
Vect caldx(BodyX b, BodyX a) {  //impulse of a on b
    Vect dx = a.c - b.c;
    Vect dv = a.v - b.v;
    double r = dx.abs();
    return G * a.m * (
           1.0 / 2.0 * dx / (r * r * r) * dt * dt +
           1.0 / 6.0 * dv / (r * r * r) * dt * dt * dt +
           1.0 / 2.0 * (dx & dv) * dx / (r * r * r * r * r) * dt * dt * dt
           );
}
void iterate(BodyX *a) {
    int i, j;
    for (i = 0; i < n; i++) {
        dv[i] = Vect(0, 0, 0);
        dx[i] = Vect(0, 0, 0);
```

```
        for (j = 0; j < n; j++) {
            if(j!=i) {
                //calculate dv from j to i
                dv[i] += caldv(a[i], a[j]);
                //calculate dx from j to i
                dx[i] += a[i].v * dt + caldx(a[i], a[j]);
            }
        }
    }

    // check and handle collision
    Vect vit, vjt;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) {
            if (((a[i].c + dx[i]) - (a[j].c + dx[j])).abs() <=
                a[i].size + a[j].size) {
                collide(a[i], a[j], vit, vjt);
                a[i].v = vit;
                a[j].v = vjt;
                dx[i] = Vect(0.0);
                dx[j] = Vect(0.0);
                dv[i] = Vect(0.0);
                dv[j] = Vect(0.0);
            }
        }

    for (i = 0; i < n; i++) {
        a[i].c += dx[i];
        a[i].v += dv[i];
    }
}
```

As we can see from above, the coding structure of this sequential algorithm contains two level loops in each iterations, each iterate through all bodies. Thus the sequential algorithm has $O(n^2)$ time complexity, and we should probably be expecting the same complexity in the parallel version without changing the algorithm.



2. Octree



An octree is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three-dimensional space by recursively subdividing it into eight octants.

In our case, every node of the octree represents all the bodies contained in the partitioned space. It has its own coordinates, velocity and mass, all of which are aggregated from its children. If the space has only one body, then the node's property is set to be the body's property. If the space doesn't have any body, then the node's property is set to 0.

Given all the bodies' information, we first build the octree to distribute the bodies to the corresponding space(nodes) and compute the aggregated property of all the nodes. Thus, when we want to calculate the force between one body A and the other bodies, we could start at the root, propagate the calculation down to the leaves and add them together. If the node represents a single body or the bodies that are far enough from body A so that they could be regarded as one aggregated body, we will calculate the force directly using the node's property and body A's property. The constant "OctreeThreshold" is used to determine whether the distance between body A and the aggregated body is large enough.

In the meantime, we could easily apply OpenMP combined parallel/worksharing construct to parallelize the calculation of the forces.

3. OpenMP

For multicore optimization, we choose to use OpenMP to optimize the computing procedure. There are multiple ways to choose from in parallelizing the sequential algorithm.

For parallel solutions, we first reorganized data structure. In previous version, we used Struct Body and relevant methods to represent body. Now for data space locality and parallel optimization, we divide large class Body into smaller data structures: Vects and floats. In this way, calculation on bodies could be transferred into parallel calculation on arrays of basic data structures.

As we mentioned in the previous section, the main part to optimize is the iteration part. We did a simple profiling over different parts in the iteration pipeline we mentioned in the sequential algorithm, we found that the most time consuming part is calculation of dv and dx .

```
-bash-4.2$ ./serial
T: 160000
n: 16000
******************iterate**********************
calculate gravity: 9952.000000 ms
calculate collide: 501.000000 ms
******************iterate**********************
calculate gravity: 9912.000000 ms
calculate collide: 502.000000 ms
******************iterate**********************
calculate gravity: 9913.000000 ms
calculate collide: 501.000000 ms
******************iterate**********************
calculate gravity: 9911.000000 ms
calculate collide: 501.000000 ms
******************iterate**********************
```

sequential algorithm profiling (input size: 16000)

As a result, we have 2 different ways of optimizing the algorithm: parallelize the inner loop or outer loop. We implemented both and in different approaches.

1. Parallel in outer loop with manual batch split:
   In this implementation, we use omp parallel to manually divide iterations of the outer loop into parallel sections in the number of the openmp threads (8). In each section, it calculates the velocity change and position shift of bodies in its section. Following is a piece of sample code:

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    int noth = omp_get_num_threads();
    int size = n / noth + 1;
    int id = omp_get_thread_num();
    int start = id * size;
    int end = min((id + 1) * size, n);

    for (i = start; i < end; i++)
    {
        dv[i] = Vect(0, 0, 0);
        dx[i] = Vect(0, 0, 0);
        for (j = 0; j < n; j++)
        {
```

```
            if(j!=i)
            {
                dv[i] += caldv(cs[i], vs[i], cs[j], vs[j], ms[j]);
                dx[i] += vs[i] * dt +
                         caldx(cs[i], vs[i], cs[j], vs[j], ms[j]);
            }
        }
    }
}
```

We didn't expect very good performance with this approach, since each thread needs to calculate impulse from all bodies for multiple times, and there's not enough time for the data caching.

2. Parallel in inner loop with parallel for:
   In this approach, we use the `omp parallel for` gragma to optimize the algorithm by parallelizing the inner loop to calculate impulse of each other body over the current body concurrently.

```
for (i = 0; i < n; i++)
{
    float vx = 0, vy = 0, vz = 0, xx = 0, xy = 0, xz = 0;

    #pragma omp parallel for \
        reduction(+: vx, vy, vz, xx, xy, xz)// shared (cs, vs, ms)
    for (j = 0; j < n; j++)
    {
        if(j!=i)
        {
            Vect ddv = caldv(cs[i], vs[i], cs[j], vs[j], ms[j]);
            Vect ddx = vs[i] * dt +
                       caldx(cs[i], vs[i], cs[j], vs[j], ms[j]);
            vx += ddv.x; vy += ddv.y; vz += ddv.z;
            xx += ddx.x; xy += ddx.y; xz += ddx.z;
        }
    }
    dv[i] = Vect(vx, vy, vz);
    dx[i] = Vect(xx, xy, xz);
}
```

We utilize the reduction function of `omp parallel for` pragma, this will automatically optimize the loop which has some form of reduction inside. One point worth mention is that we are defining a struct `Vect` in the code for our convenience, while the reduction pragma cannot be used upon struts. As a result, we have to break it into floats reduce over these variables.

To optimize the performance, we tried the shared pragma on the data sources (cs, vc, ms). Yet we failed to observe any improvement in performance. It's possible that the `omp parallel for` pragma already

shared these data by syntax analysis to determine that they are only read in the process.

4. CUDA

For manycore optimization, we choose to use CUDA to optimize the computing procedure. As mentioned in OpenMP section, in our case, we mainly optimized the part of calculation of dv and dx.

We divided inner loop of calculation into GRID_SIZE partition. Main calculation is operated on the GPU parallelly.

We defined NUM_THREADS_PER_BLOCK as 1024 for best performance.

```
for (i = 0; i < n; i++)
{
    cal_gravity_kernel<<<GRID_SIZE, NUM_THREADS_PER_BLOCK>>>
        (cs_d, vs_d, ms_d, sizes_d, dv_d, dx_d, n, GRID_SIZE, i);
}
```

```
void
cal_gravity_kernel(Vect * cs_d,Vect * vs_d,float* ms_d,float* sizes_d,Vect *
dv_d,Vect *  dx_d,int n,int GRID_SIZE, int i) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    if(index < n && i != index) {
        dv_d[i] +=
            caldv(cs_d[i], vs_d[i], cs_d[index], vs_d[index], ms_d[index]);
        dx_d[i] += vs_d[i] * dt +
            caldx(cs_d[i], vs_d[i], cs_d[index], vs_d[index], ms_d[index]);
    }
}
```

In this version, we got a huge improvement on calculation time and then we noticed that some global data has been accessed for multiple times so we restore them into shared memory. Note that we should use __syncthreads() to ensure synchronization for all threads. Following is the final sample code.

```
void
cal_gravity_kernel(Vect * cs_d,Vect * vs_d,float* ms_d,float* sizes_d,Vect *
dv_d,Vect *  dx_d,int n,int GRID_SIZE, int i) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    __shared__ Vect dv_s, dx_s, cs_d_i, vs_d_i;
    __shared__ float dt_s;
    dv_s = Vect(0.0f, 0.0f, 0.0f);
    dx_s = Vect(0.0f, 0.0f, 0.0f);
    cs_d_i = cs_d[i];
    vs_d_i = vs_d[i];
    dt_s = dt;
    __syncthreads();
    if(index < n && i != index) {
        dv_s +=
```

```
            caldv(cs_d_i, vs_d_i, cs_d[index], vs_d[index], ms_d[index]);
        dx_s += vs_d_i * dt_s +
            caldx(cs_d_i, vs_d_i, cs_d[index], vs_d[index], ms_d[index]);
    }

    __syncthreads(); // wait for ending of all threads
    dv_d[i] = dv_s;
    dx_d[i] = dx_s;
}
```

Analysis of Results

Our hardware resources' specifications are as follows.

Manycore hardware resource

GPU Name:	NVIDIA GeForce GTX 1080
Number of threads per block	1024
Number of threads in a warp	32
Max no. of threads per SM	2048
Max no. of warps per block	32
Max no. of warps per SM	64
Max no. of FLOP per thread per cycle	128
Peak performance	8873 GFLOPS
Total amount of L2 cache	2097152 bytes

Multicore hardware resource

Number of physical CPU :	4
Number of virtual CPU :	8
Model name :	Intel(R) Core(TM) i7-4785T CPU @ 2.20GHz
CPU / MHz :	2799.929

L1d cache :	32K
L1i cache :	32K
L2 cache :	256K
L3 cache :	8192K

1. Sequential Result

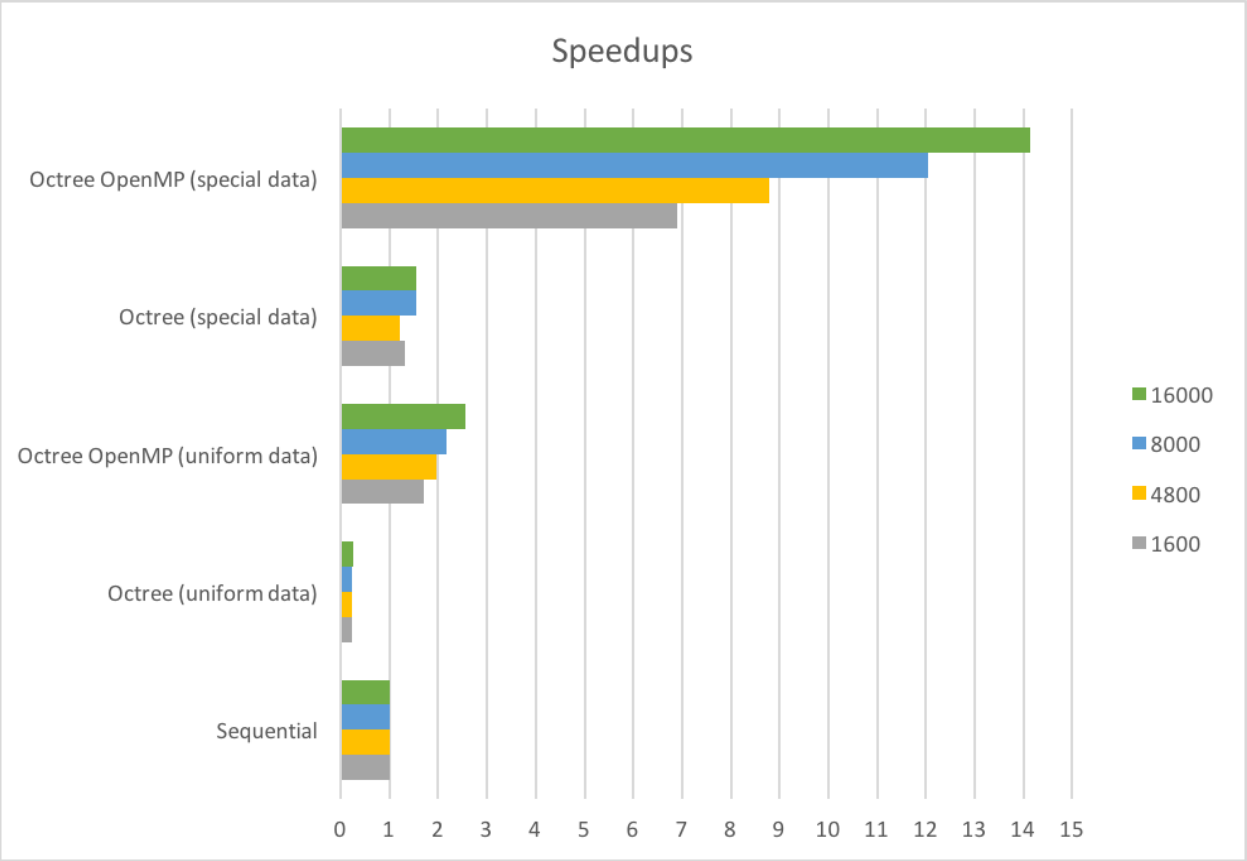
Number of Bodies	Sequential Version (uniform data) (ms)	Sequential Version (special data) (ms)
1600	113	122.5
4800	864	865.5
8000	2431	2452.25
16000	10423	10529.25

1. Result optimized by Octree

Number of Bodies	Octree (uniform data) (ms)	Octree OpenMP (uniform data) (ms)	Octree (special data) (ms)	Octree OpenMP (special data) (ms)
1600	486.25	66	93.25	17.75
4800	3812.25	441	719.75	98.5
8000	10529	1126.25	1572.25	203.5
16000	38849.75	4089.5	6813.75	744.5

Number of Bodies	Octree (uniform data)	Octree OpenMP	Octree (special data)	Octree OpenMP

	(speedup)	(uniform data) (speedup)	(speedup)	(special data) (speedup)
1600	0.23	1.71	1.31	6.90
4800	0.23	1.96	1.20	8.79
8000	0.23	2.16	1.56	12.05
16000	0.27	2.55	1.55	14.14



We can see in the chart that the OpenMP version of Octree is around 4-9 times faster than the initial version. If OpenMP is not applied, the performance of Octree is much worse than the Sequential version when dealing with uniform data; And when dealing with special data, its performance is only slightly better than the Sequential version.

And the performance of special data is around 3-6 times better than the uniform data. By uniform, we mean the bodies are evenly distributed in the space. By special, we mean the bodies are divided into 8 groups, where each group is near the vertexes of one cube and the bodies in each group are evenly distributed around the vertexes. By

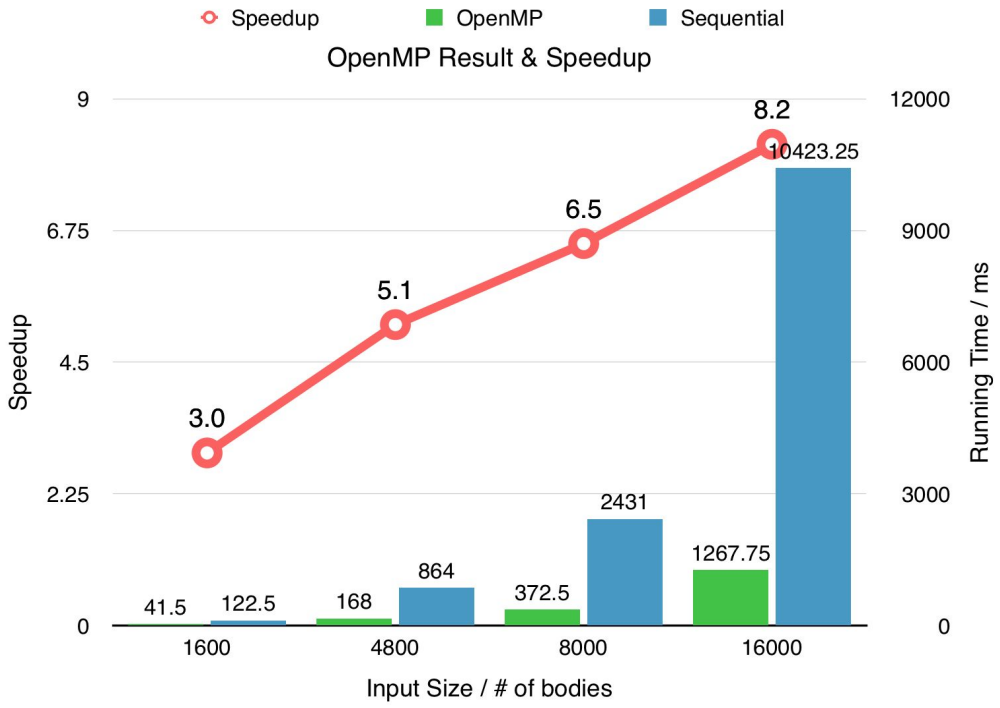
doing this, we can make sure the distance between every two groups is large enough so that we have the chance to regard each group as one aggregated body to accelerate calculation. With special data, the performances of two versions of Octree are both better than the Sequential one.

As the number of bodies increases, the speedup of OpenMP versions gradually increases. The reason is that OpenMP has sequential overhead. When the portion of parallel computation increases, the overall performance gets better.

In conclusion, OpenMP contributes a large portion of the speedups and the data distribution really matters for Octree optimization.

2. Multicore Result

Number of Bodies	OpenMP (ms)	Speed Up (Compared to Sequential Version)
1600	41.5	2.95x
4800	168	5.14x
8000	372.5	6.53x
16000	1267.75	8.22x

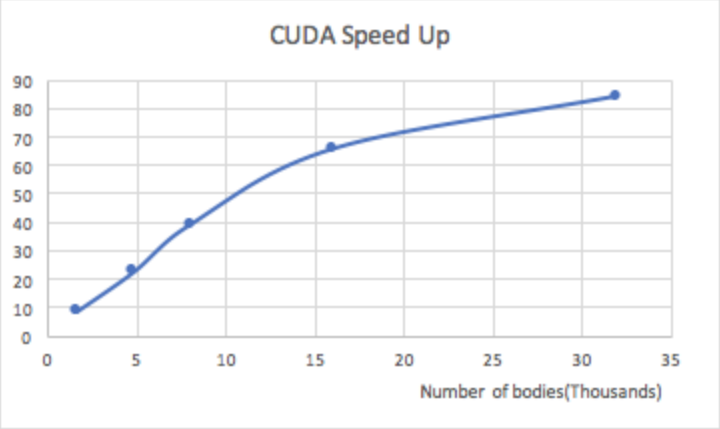


As we are using Intel Core i7 CPU with 8 cores, we should be expecting up to 8x speed up in its performance. As shown in the graph, as the data size increases, the rate of speedup increased from around 3x to 8x, which is an expected observation.

It's reasonable that the speedup are not able to reach 8x in smaller sized input, since the parallel implementation has multiple level of overhead, which would take less and less portion of all running time while input size grows. Finally, the overhead can be ignored and the performance will be close to the expected speedup.

3. Manycore Result

Number of Bodies	Cuda (ms)	Speed Up (Compared to Sequential Version)
1600	13	8.69x
4800	38.25	22.59x
8000	62	39.21x
16000	158.75	65.65x
32000	456	84.11x
40000	641	96.72x

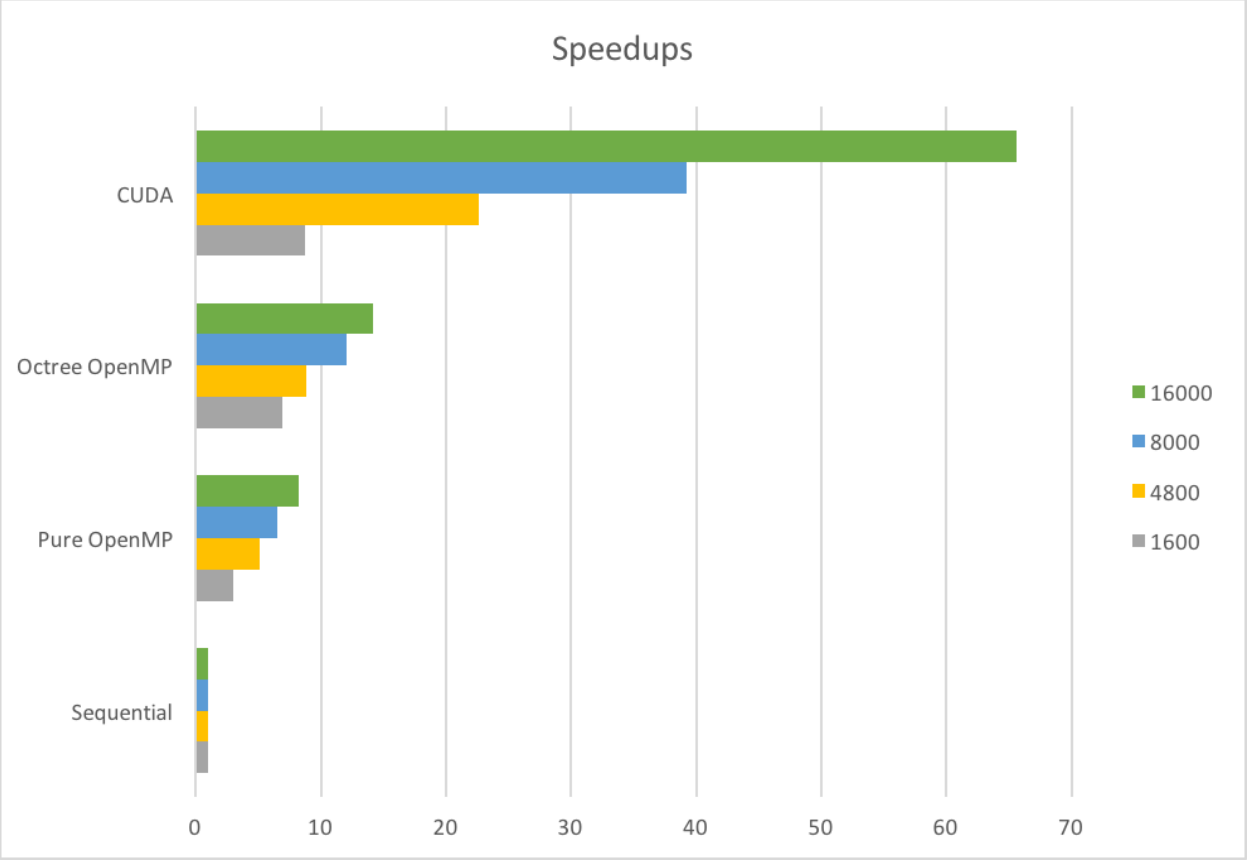


We found that the speed up of CUDA method is almost 96x for large-scale data. As we are using NVIDIA GeForce GTX 1080, it can run a significantly larger number of operations per second than the CPU.

As the number of data increases, the speedup gradually increases because of the overhead of manycore calculation. We may not observer the rooftop speedup because it may consume much time.

Overall, CUDA method gets the best performance based on our machine resources.

Conclusion



The CUDA version wins the first prize! The follows are Octree OpenMP version and Pure OpenMP version. The CUDA version has the highest level of parallelization and the Octree version demonstrates the ability to accelerate the calculation by regarding a group of bodies as one single aggregated body.

As the number of bodies increases, the speedup of all versions gradually increases. The reason is that OpenMP and CUDA has sequential overhead. When the portion of parallel computation increases, the overall performance gets better.

Take Home Messages

First, we realized that manycore framework optimization has great advantage when dealing with regulated calculation with little or moderate amount of data dependency. We are able to dig up huge performance (as much as 100+ times speedup) solving this problem with CUDA.

Second, we leant that some problems or algorithms are very data-sensitive. We need to thoroughly investigate the possible distribution and form of input data to determine if we have good performance in all cases. This is a basic concept in algorithm design, and it applies for parallel algorithm design as well.

Reference

[1] Frasheri, Neki, and Betim Çiço. "Scalability of gravity inversion with OpenMP and MPI in parallel processing." *ICT Innovations 2012*. Springer Berlin Heidelberg, 2013. 345-352.

[2] Bédorf, Jeroen, and Simon Portegies Zwart. "Parallel gravity: from embarrassingly parallel to hierarchical." Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date. ACM, 2012.

[3] Vasiliadis, Giorgos, and Sotiris Ioannidis. "Gravity: a massively parallel antivirus engine." International Workshop on Recent Advances in Intrusion Detection. Springer Berlin Heidelberg, 2010.

[4] Schwartz, Stephen R., Derek C. Richardson, and Patrick Michel. "An implementation of the soft-sphere discrete element method in a high-performance parallel gravity tree-code." *Granular Matter* 14.3 (2012): 363.

[5] Moore, Matthew, and Jane Wilhelms. "Collision detection and response for computer animation." *ACM Siggraph Computer Graphics*. Vol. 22. No. 4. ACM, 1988.