Spring 2019 ME759 Final Project Report
University of Wisconsin-Madison

Default Final Project 2: Collision Detection

Wen-Fu Lee
Yuan-Ting Hsieh

May 7, 2019

Abstract

This project trying to do collision detection between a mesh and spheres, where the mesh is a closed shape and all spheres have the same radius. We are utilizing what we've learned in this class, including CUDA programming with shared memory and thrust library. We also use cuda-memcheck and nvprof to profiling our application. We are able to reduce our runtime from 36.5 minutes using brute-force on CPU to 300 ms using the bucket algorithm and shared memory on GPU, which is a 1500x gain.
Github link: https://git.cae.wisc.edu/git/me759-wlee256/

Contents

1. General information

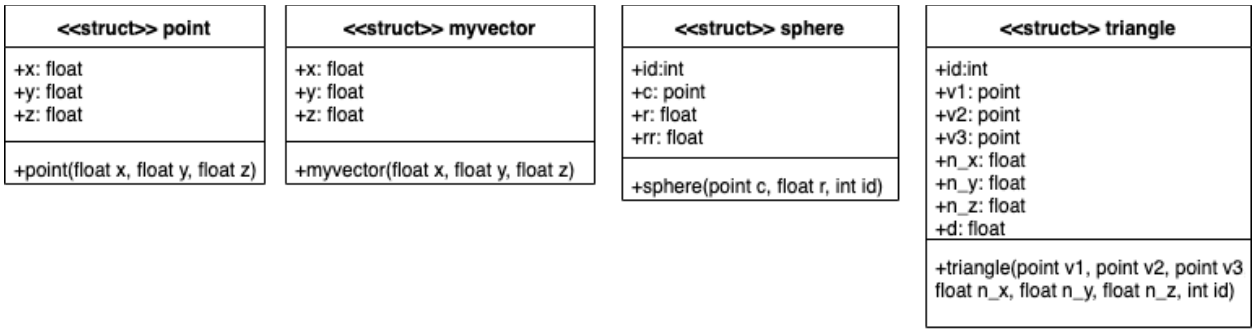
- Computer Sciences
- MS
- Wen-Fu Lee (Team Leader)
- Yuan-Ting Hsieh (Team member)
- We don't have an advisor.

2. Problem statement

We want to work on Default Final Project 2: Collision Detection, which is to implement a parallel algorithm for detecting collisions between a triangle mesh and a large collection of spheres. In practice, collision detection is a computationally expensive part of a simulation pipeline and it represents a prime target for well-tailored algorithms and/or parallel computing.

3. Solution description

3.0 Parser and Data Structure



We design several custom structures to represent our data more intuitively. We have point and myvector representing points and vectors in 3D space, respectively. We also have sphere and triangle structs. The struct UML diagram is shown above. To parse the mesh from Wavefront obj file, we use the tiny object loader in [4]. With triangulate set to true, every surface is going to be represented in triangles. To parse spheres, just parse it like a regular CSV file. We parse the files into vectors of spheres and triangles that we can use later on.

3.1 Collision detection

To detect the collision of a triangle and a sphere, the detection flow in [1] is adopted. It has 4 steps. Only the previous steps pass, and the next step will be executed. If one of the steps isn't satisfied, the detection halts and means there is no collision in this pair. In the following steps,

we assume that each sphere is represented by a 3-dimensional point C, its radius R, and R^2 (for speed up reasons). Each triangle T is defined by its vertices ($T.P_1$, $T.P_2$, $T.P_3$), its normal vector $T.N=(\alpha,\beta,\gamma)$ and a constant $T.\delta$ (from the triangle's plane equation $\alpha x + \beta y + \gamma z + \delta = 0$).

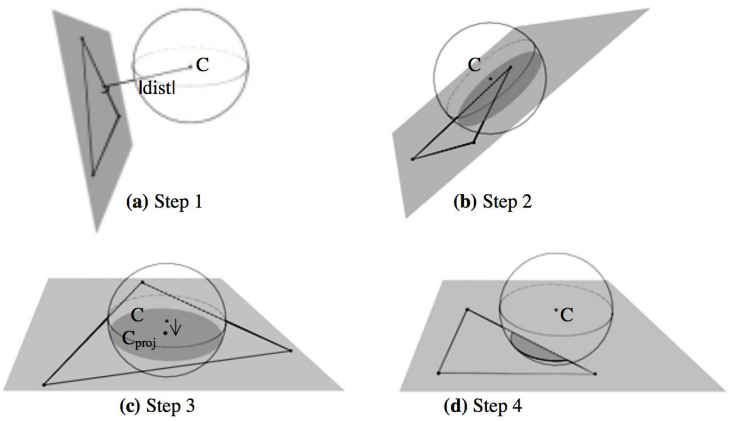


Figure 1. Sphere/triangle intersection test

Step 1. Check whether the triangle's plane intersects with the sphere. If not, there is no sphere-triangle intersection(Fig. 1a). Else go to step 2.

- This is done by calculating the plane's distance *dist* from the sphere's center (using the plane equation: $dist = T.N \cdot C + T.\delta$) and comparing $abs(dist)$ to R.

Step 2. Check whether any of the triangle vertices is inside the sphere. If yes, the sphere and the triangle intersect (Fig. 1b). Else go to step 3.

- The test is done by calculating each vertex's distance d from the sphere's center and comparing d^2 to R^2.

Step 3. Project the sphere onto the triangle's plane : $C_{proj} = C - dist \cdot T.N$. Check if the projected center C_{proj} lies inside the triangle [2]. If yes, there is an intersection (Fig. 1c). Else go to step 4.

Step 4. Check whether the sphere intersects with a triangle edge. If yes, the sphere and the triangle intersect (Fig. 1d). Else there is no intersection.

- The way we analyze the sphere/edge intersection test is in Fig. 4. Below, we use bold symbols for position vectors and plain symbols for points.
- Given a line segment defined by two points P_1, P_2 and a sphere defined by its center C and radius R, we want to determine whether the two objects intersect. If the line through C which is perpendicular to P_1P_2, intersects with P_1P_2 at Q, then our problem is equivalent to determining whether Q lies between P_1 and P_2 and comparing R^2 with the square of the distance d between C and Q.

- Let N be P_2-P_1. Using the parametric equation of the line segment P_1P_2,
$$Q = P_1 + t \cdot (P_2 - P_1) = P_1 + t \cdot N$$
- The parameter t for point Q can be calculated by:
$$t = (N \cdot C - N \cdot P_1) / (N \cdot N)$$
- Let the square of the distance between Q and C be d^2. If $d^2 > R^2$, there is no intersection. If $d^2 = R^2$ the line is tangential to the circle and the contact point Q is between P_1 and P_2 if $0 \leq t \leq 1$.
- If $d^2 < R^2$, the line defined by P_1 and P_2 intersects with the sphere. If $0 \leq t \leq 1$ then Q lies between P_1 and P_2 and the line segment P_1P_2 intersects with the sphere. Otherwise, an intersection occurs if the closest endpoint to Q lies inside the sphere. The closest point is P_1 if $t < 0$ or P_2 if $t > 1$. However, we have already excluded that possibility in Step 2. Therefore if $t < 0$ or $t > 1$, P_1P_2 does not intersect with the sphere.

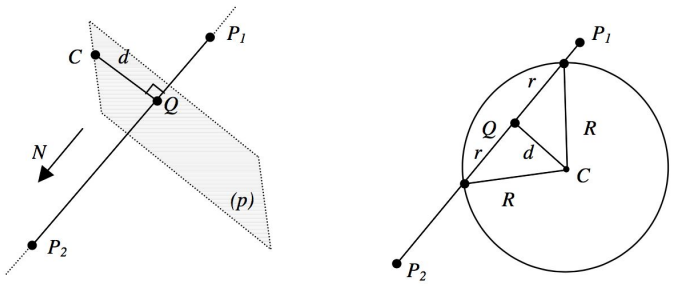


Figure 2. Line/sphere intersection.

3.2 Methods

3.2.1. Brute-force method

In order to be familiar with this topic and get a baseline for a comparison reference, we start implementing in a brute-force way first. To be specific, we not only implement a brute-force method using CPU but also try to improve it using GPU and shared memory.

3.2.1.1 Sequential processing

Here, sequential processing means that we use only CPU resources. Basically, this can be implemented by using just two for loops. One is to iterate all the triangles and the other is to iterate all the spheres. By doing this, we can get all the combinations of the pairs of a triangle and a sphere. Then, for each pair, we apply the collision detection in 3.1 to detect if there is a collision or not.

One more thing to note is that we cannot directly allocate a memory space with the size of all the pair combinations to store the collision results for each pair since the number of total pairs could be very large and the memory is not feasible. For example, the sample files provided by TAs have 62,976 triangles and 1,703,244 spheres in sample_mesh.obj and sample_spheres.csv,

respectively. Instead, we use two vector structures in C++ to *push_back* a sphere and a triangle's IDs only when they have a collision. In this way, a lot of memory spaces is saved since the number of collisions is relatively low compared to that of the total pairs.

3.2.1.2 Parallel processing

The brute-force method in sequential processing is easily implemented but really slow. So, we are curious about how this processing can be accelerated by using GPU parallel processing. Given the sample files provide by TAs, it is feasible to save all the spheres and triangles in the GPU memory at the same time, but the same issue arises if we want to store all the pairs' collision results, as stated in 3.2.1.1. Thus, we split the space of all the sphere-triangle pairs, as in Fig. 3, into multiple grids, each of which is processed by one kernel. In this way, the brute-force method in parallel processing can be implemented without any memory issue. Note that since the number of spheres or triangles may not be divisible by the grid size (10,000 x 10,000 in our case), the number of total threads in use in each grid may be different (e.g., the gray grids in Fig. 3). This is the corner case we need to deal with carefully.

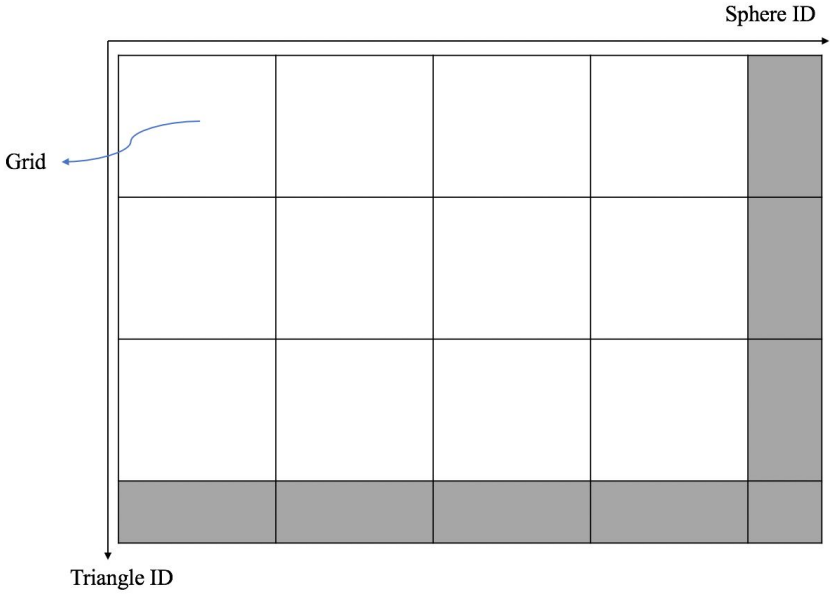


Figure 3. Each grid is processed by one kernel.

3.2.1.3 Parallel processing with shared memory

Shared memory is a special entity. It is on-chip, so it is much faster than local and global memory. In fact, shared memory latency is roughly 100x lower than uncached global memory latency (provided that there are no bank conflicts between the threads). Moreover, shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. Thus, using shared memory well can further accelerate GPU parallel processing. To be specific, the parallel processing with shared memory is implemented on top of the structure in 3.2.1.2, as shown in Fig. 4. The size of shared memory in each block is defined as

$$shaMemSize = blockIdx.x * sizeof(sphere) + blockIdx.y * sizeof(triangle)$$

According to the above equation, we know that only (*blockIdx.x* + *blockIdx.y*) threads are involved in moving triangles and spheres from global memory to shared memory for each thread block. Again, note that since the grid size may not be divisible by the block size (16 x 16 in our case), the number of total threads in use in each block may be different (e.g., the gray grids in Fig. 4). This is the corner case we need to deal with carefully.

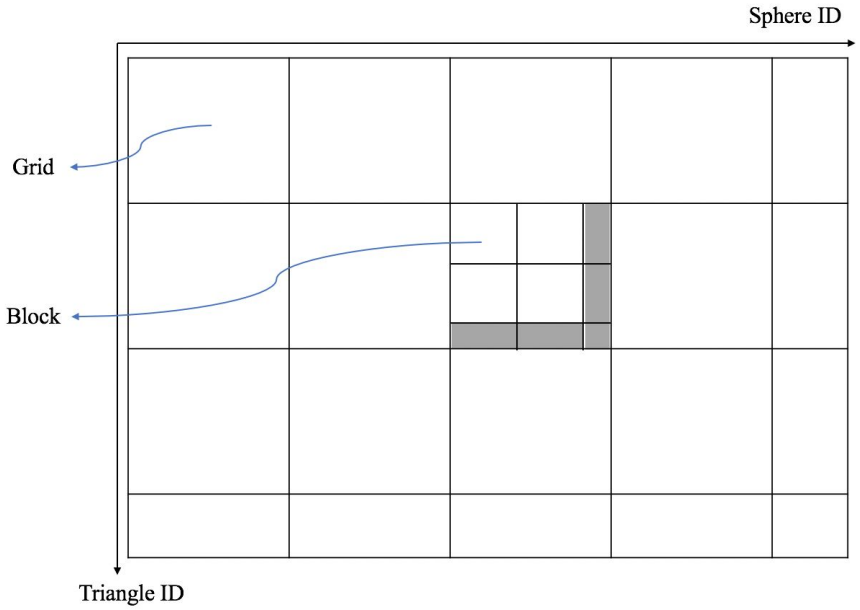


Figure 4. Each block inside a grid accesses the same shared memory.

3.2.2. Bucket

To speed up the collision detection process, we adopt some of the ideas from [3] and decide to split triangles and spheres into "buckets". And inside each bucket, we do the brute force all-pair collision detection. When parsing the spheres and triangles, we are finding the bounding box of our total space. Once we found that, we choose a bucket width of 4 * radius of a sphere to split the entire space into small blocks. Note that a sphere or triangle could cross multiple buckets, so we find a bounding box first for each object then use the bounding box to check which buckets this object intersects. So, in the end, we will have an array whose length is total buckets while each entry is also an array contains all the objects inside the bucket.

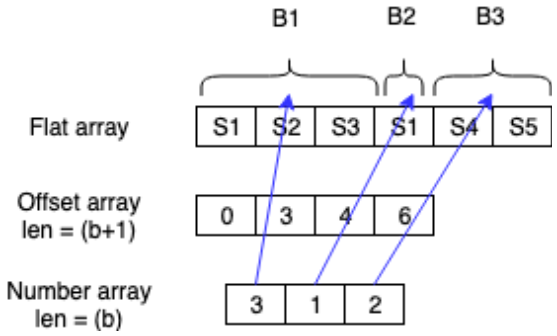
3.2.2.1 Sequential processing

For each bucket, enumerating all-pairs and check if they collide using CPU.

3.2.2.2 Parallel processing

As we discussed above, we now have an array of arrays of triangles/spheres. Inspired by the idea of stage 7 ~ 9 from [3]. Our algorithm then proceeds in the following steps.

Step 1. Serialize the array of arrays of triangles/spheres to just a single flat array of triangles/spheres. And we also need another array specifying the offset for each bucket. That is the starting position in the flat object array that this bucket should deal with. So subtract next offset with current offset we could get the number of triangles/spheres that belong to this bucket. So the number array is calculated from the difference of offset array. The concept is depicted below.



Step 2. For each bucket, we launch a GPU thread to handle that. We calculate the number of collisions for each bucket.

Step 3. Use thrust library to do an inclusive scan to get the total number of collisions

Step 4. For each bucket, we now know the collision offset and we know the total number of collisions for each and all bucket. We could use this information to write out collision information, which in our case is just ID pairs. Note that in here we still launch a GPU thread for each bucket.

3.2.2.3 Parallel processing with shared memory

When calculating the collision number and get collision information, the sphere offsets array and triangle offsets array are used consecutively by each GPU threads. That means each of them is load twice, we could use shared memory so that they only need to be load once from device memory.

4. Overview of results. Demonstration of your project

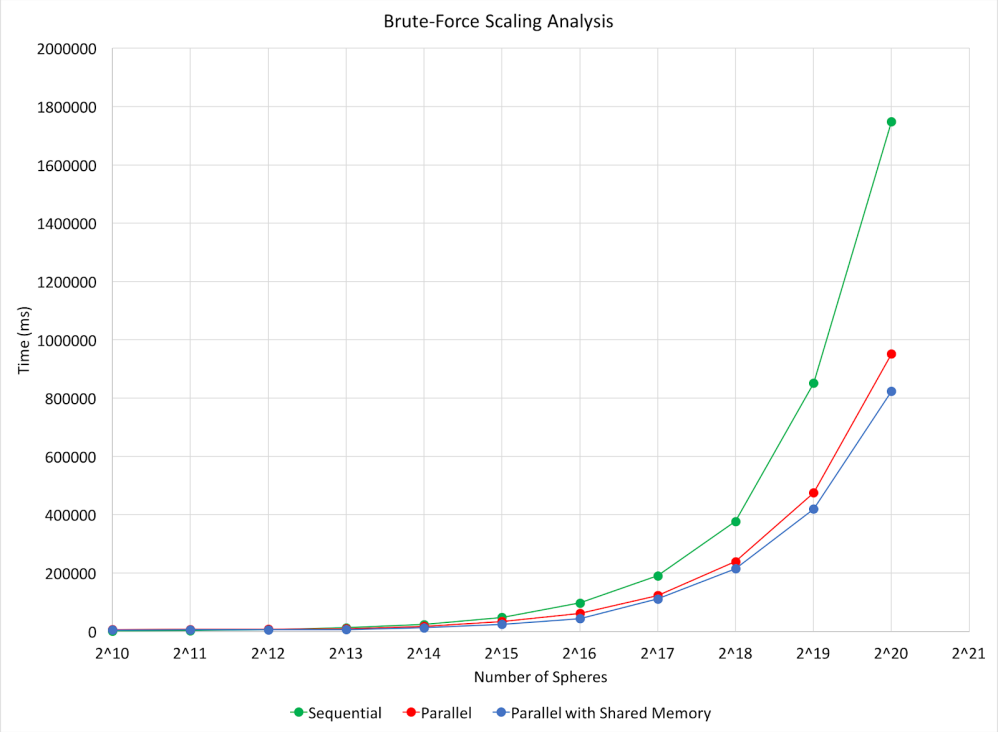
In both the following scaling analysis, we use the same sample_mesh.obj provided by TAs and set the sphere's radius to 3.0. We want to compare the efficiency of different methods under different numbers of spheres.

4.1. Brute-force method

The scaling analysis below shows the performance of different brute-force methods. As expected, the efficiency of them is as follows.

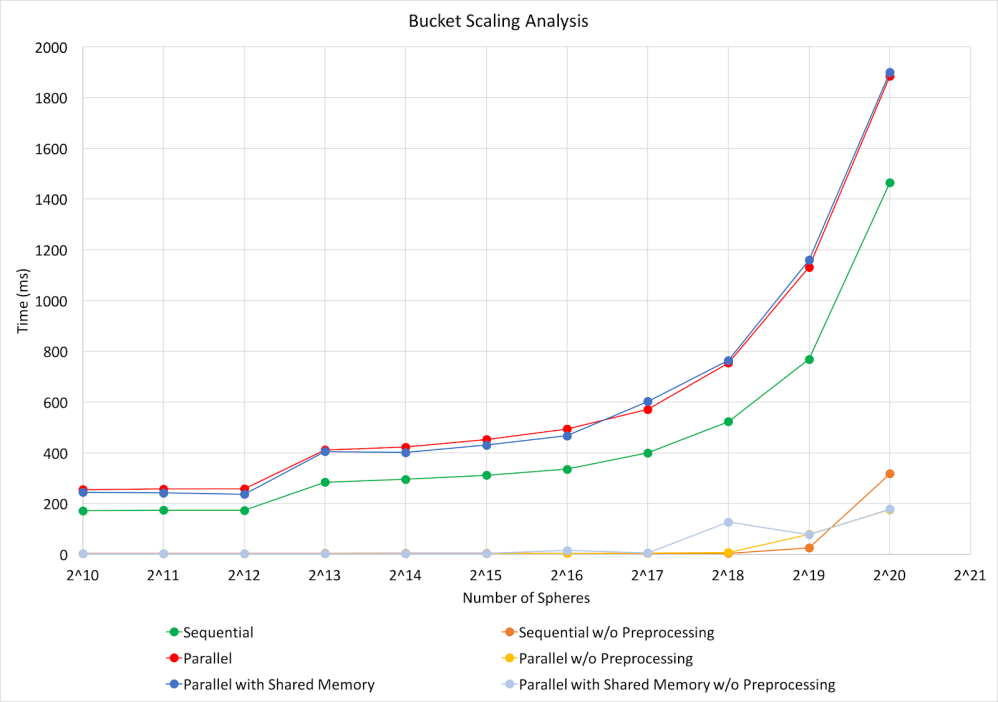
Brute-Force Parallel with Shared Memory > Brute-Force Parallel > Brute-Force Sequential

For example, when the number of spheres is 2^{20}, Brute-Force Parallel and Brute-Force Parallel with Shared Memory are 1.83 and 2.12 times faster than Brute-Force Sequential. This proves the idea again that always think about how to parallelize your algorithm, and use shared memory if possible.



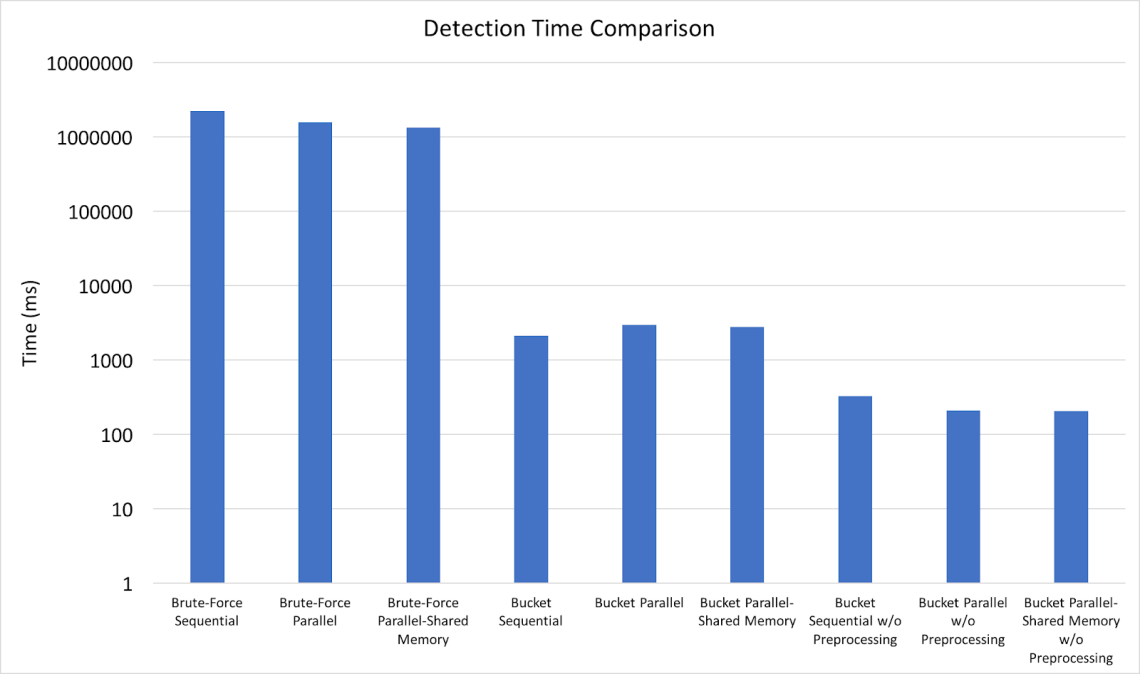
4.2. Bucket method

The scaling analysis of different bucket approach is depicted below. This approach is faster than the brute-force ones. Even simple CPU sequential implementation can have a run time of 1465 ms when the number of spheres is 2^{20}, which is around 1000x faster. The run time of the GPU version is 1884 ms which is slower than sequential implementation, This is because it needs an extra step of preprocessing than the sequential code. And right now that part is only run on CPU which takes 565 ms. If we exclude all preprocess time then the run time of parallel code is 178 ms and the runtime of sequential code is 322.35 ms, which is a 2x speed up. This speedup can even be more if we parallelize the preprocessing stage as in [3]. In total, we improve around 1000x compare to brute force. Note that using shared memory in here is not gaining. Maybe it is because it only reduces one load time which is not significant in here.



4.3. Summary of TA's sample example

In addition to the above scaling analysis, we also run the complete sample files provided by TA: sample_mesh.obj and sample_spheres.csv, and set the radius of spheres to 3.0. The following chart shows the comparison among our 6 methods. Note that they all get the same collision number: 39,709.



5. Deliverables

- This report is uploaded in Canvas.
- Our git repo folder is called Final_Project instead of FinalProject
- The git repo contains many subfolders, each of which corresponds to the methods we have above and is organized as follows:

Folder name	Description
brute_force_seq	brute-force sequential processing
brute_force	brute-force parallel processing
brute_force_shamem	brute-force parallel processing with shared memory
bucket_seq	bucket sequential processing
bucket	bucket parallel processing
bucket_shamem	bucket parallel processing with shared memory

- For each subfolder above, there is a CMakeLists.txt and a main.cu file. To compile and run examples, type "sbatch main.sh", which already includes "cmake .; make". Note that all the sample data are not put in git because they are too large. **So, you may need to manually upload those test files to our git folders and modify the sample file paths in main.sh.**

6. Conclusions and future work

By using the bucket algorithm, we get a 1000x speed up. Further utilizing CUDA and GPU computing, we are able to get additional 8x speed up. As we check what can be further improved using nvprof, we saw that around 20% of the time is spending on copying memory from host to device. This can be improved if we also do the preprocessing stage (related to stage 1 ~ 6 in [3]) in parallel in GPU. That way we can reduce the preprocessing time and also copying fewer data from the host to the device. We do also investigate on the constant memory as mentioned in [3] and our proposal. But it looks like there is no dynamic allocation on that part, and we don't want to make the bucket size to be fixed (right now, they are determined by using width = 4 * the radius of the sphere). But if we were to fix the bucket size, then *sphereOffset* and *triangleOffset* can be put inside the constant memory which will further reduce runtime.

References

[1] E.-A. Karabassi, G. Papaioannou, T. Theoharis, and A. Boehm. Intersection test for collision detection in particle systems. J. Graph. Tools, 4(1):25–37, 1999.
[2]https://gamedev.stackexchange.com/questions/23743/whats-the-most-efficient-way-to-find-barycentric-coordinates
[3] H. Mazhar, T. Heyn, and D. Negrut. A scalable parallel method for large collision detection problems. Multibody System Dynamics, 26:37–55, 2011.
[4] https://github.com/syoyo/tinyobjloader