N-body Dynamic Collision Detection Parallelism with Cuda

Hao Ding, Tianxiang Gao

SUMMARY

We implemented V-collide collision detection in CUDA on the GPU and compared the performance of the parallelized version with the sequential version on CPU. We achieved 10x speed up for simple cases and 100x speed up for complex cases.

BACKGROUND

Collision detection is a mechanism determining if several objects overlap with each other in a given time or not. The detection is commonly applied in a sequence of frames, aiming at a large number of objects for simulation purposes. For this project, we would like to leverage the feature of the independence of objects to conduct frame by frame parallelization over CUDA. Based on the background research we have conducted, we started from the V-Collide algorithm, which performs well for the measurement of arbitrary-shaped objects.

For this project, we parallelize our code and measure the performance on the GHC machine with CUDA.

1. Input and Output

An arbitrary shaped object is represented as a set of linked triangles. The movement of the object is represented by a 4x4 transformation matrix, applied to each of the triangles. Thus, there are two input for the functionality:

1. Triangles.

    Each triangle is determined by three points represented in X, Y, Z dimensions separately. The first line of the file describes how many triangles each object contains, helping the program understand how to group triangles as an object.

2. Transformation matrix.

    For our test case, the shape of the object is not changed, only the position of the object changed from time to time. This helps us to test the correctness of the program and to simplify the situations. However, the cases involving scaling and transforming of the object can also be handled.

    At the same time, the collision in the current timestamp will affect the object's movement for the following timestamps. This requires related modifications of object positioning apart from the fed-in transformation matrix.

The purpose of the project is to detect the collision of objects from time to time. Accordingly, we use standard output to print the pairs of two colliding objects for each timeframe.

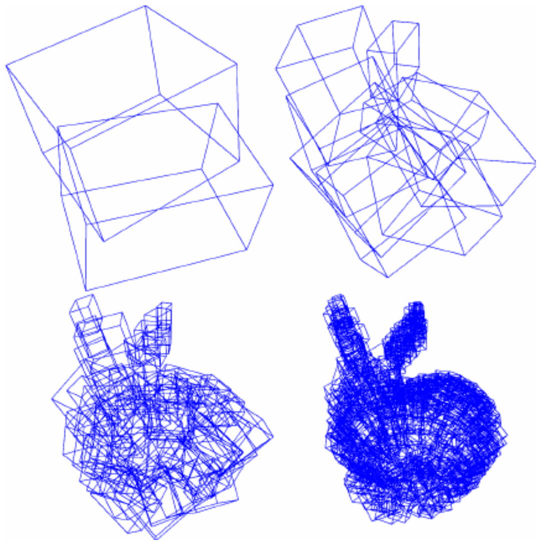
2. Logic

a. Broad Phase or the N-body

In this phase, we will assign an AABB box for each object. The AABB box is a rectangle box that contains the entire object. The length, width and height of the AABB box is determined by the longest distance of any 2 points in that particular dimension. Having an AABB test before a specific test can significantly reduce the time of comparing. The data structure in this phase for the sequential code is linked lists. One linked list consists of the sorted sequence of objects in that particular dimension. On object movement, the object node needed to be deleted and reinserted into the linked list.

b. Narrow Phase or OBB

For this phase, we need to find possible colliding triangles (minimal unit to render images). Because comparing every 2 triangles is very time consuming, the algorithm uses a binary tree hierarchy approach. The root node of the binary tree is a bounding box for the entire object. After that, each node has 0, 1 or 2 children. The algorithm does it work to ensure that the triangles are allocated equally for the 2 children. For this phase, the algorithm can find possible colliding triangles from a top to bottom approach. If 2 triangles are not likely to collide, further detailed tests would not be done.



https://www.researchgate.net/figure/Collision-detection-using-OBB-tree-in-2D_fig2_2368 34317

c. Mathematical Calculations

The last phase is a math and calculation based test that determines if the 2 triangles collide or not.

3. Pre-Analysis for Parallelization

The main idea of using parallelization is that all of the three phases mentioned in the last section are computationally expensive, and the movement of objects are independent from each other. Also, the determination mostly depends on the same mathematical calculations with different position data. Thus, this is a good match for using CUDA to leverage this independence and similarity.

However, dependences still exist. The instruction order of the three phases is required to be strictly preserved so that the next phase can make calculations according to the previous phase's intermediate results. Also, the parallelization over frames may not be possible, given that the real transformation matrix both depends on the input and the result of collision detection from the last frame.

As for locality, since there are not frequent ping-pong between objects, we don't expect the data locality to be bad.

The original code is not amenable to SIMD instructions, because of the usage of the entry data structures. After our modification, SIMD instructions are possible.

PARALLELIZATION APPROACH

To parallel this collision detection algorithm with cuda on the GHC machine, we design a 3 step parallel algorithm that parallel the broad phase, the narrow phase, and the initial phase that build the OBB model.

Sequential to Parallel

Broad Phase: Linked List to Sorted Array

The initial implementation in sequential code for the broad phase was implemented with a linked list. Implementing the parallel code in linked lists means that we need to add additional locks or write locked free linked lists to handle this problem. Furthermore, because we design our own data structure for each node, it is hard to use the atomicCAS provided by CUDA. We also noticed that the CUDA implementation of locks often ran into deadlock. Thus, we change the data structure to a sorted array. To find the collision between 2 objects, we first sort the array based on values for the three dimensions. Then for each object, we compare its value to its adjacent value until its higher value at that particular dimension is smaller than the lower value of the next object. After that, we find the objects that have overlaps in 3 dimensions and add them to our possible - collision - pair. We proposed 2 sorting algorithms to handle different cases: merge sort and bubble sort. At first, we seeked other sorting algorithms that would achieve good performance in parallelism. For example, radix sort. However, those algorithms are hard to implement with our own data structure that has multiple floating point variables. For parallelized merge sort, we parallel the "merge" phase. The idea is shown in the graph:

phase 1	1 2	3 4	5 6	7 8
phase 2	1 2 3 4		5 6 7 8	
phase 3	1 2 3 4 5 6 7 8			

However, the problem with our cuda merge sort implementation is that its last step, merge 2 big list steps, is sequential. Thus, we proposed the parallelized bubble sort in cuda. The idea is the following. For each step,

1. We compare each value of the array with its next value. If the next value is greater than the current value, we mark the buffer of this value as 1

2. If we find out no buffer is marked, we are done.
3. Swap this value and next value only if the buffer of the next value is 1 (e.g. it will not be swapped out and access by other thread)

This code would run O(N) steps if the original sequence is in really bad shape. For example, it takes roughly 60 steps for 32 objects if the object is in complete reverse order. However, the trick in collision detection is that the objects do not transpass each other. If the movement of the object is not too extreme (all objects move extremely fast all over the place), it would have great performance. We propose two sorting algorithms for different situations. If the movement of objects is very extreme, then merge sort is better, otherwise bubble sort is better.

Narrow and OBB Phase: Avoid Recursion

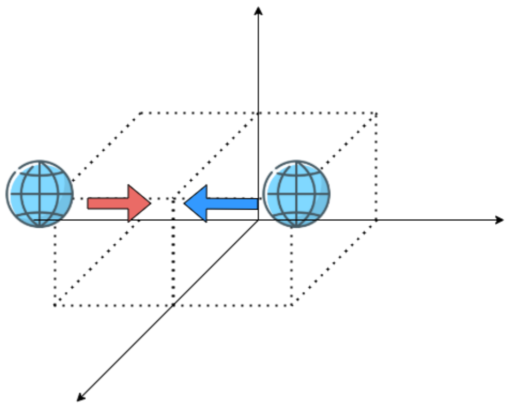
For this phase, we parallel over the possible collision pairs we generated in the last phase. In the original code, the going through of the hierarchy structure is done by a recursive function and involves a large amount of interacting with the global variable. However, both the recursive function and the global variable are very hard to parallel: the global variable will change over time in parallel function, and CUDA could not handle recursion so deep, so it quickly went into segmentation fault. Furthermore, cuda has no built- in stack and queue operation. To solve these two problems, we need to change the data structure.

First of all, we need to flatten the original recursive code. To do so, I create a cuda container data structure that wraps all things needed to do the recursion. Then, I use array structure to use as a substitute for the stack. Instead of going to the next recursion, we added the next recursion to our created cuda container buffer and added the allocation count 1. Meanwhile, when we step into a box, we do the operation and add the used count 1. We know that the 2 objects do not collide if the used count is larger than the allocation count. In order to successfully run the CUDA function, we need to find a sweet spot of the buffer size: if the buffer size is too large, then the CUDA would run the memory out. If it is too small, we have to realloc while doing the recursion which will add additional work. On the other hand, we reduce the use of global variables by adding additional buffers and rearranging the code. For example, in the original code, one of the early collision detection stopping criteria is written as global. I rearrange the functions that use the global to make things work.

Mapping

As described in previous sessions, we map objects to each thread of the blocks to leverage the independence of object features.

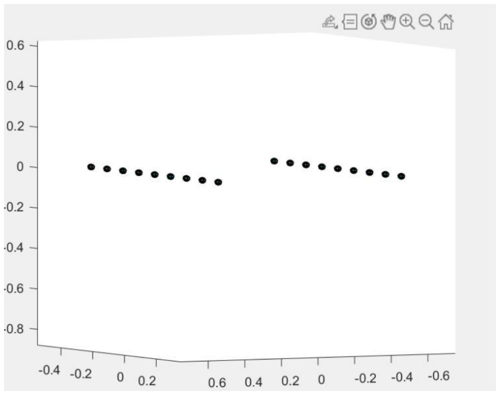
RESULTS



Test Case

Correctness

To verify the correctness of the program, we use a simple case with 2 spheres placed in the same x positions and z positions with different y positions. For the transformation, they move towards each other, and should collide in the middle of the path. After the validation of the simplest case, we scale up the number of spheres to 1024. 512 pairs of balls moved towards each other in this test case.



Performance and Scalability

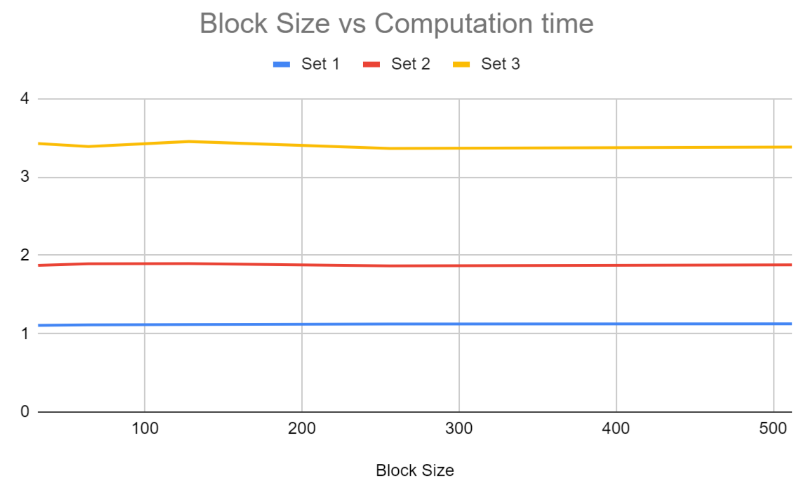
After the correctness checking, we want to measure the performance with random data. Thus, the following cases are placed in random positions for the first frame, and moved in a random manner like Brownian Movement.

a. ***Complexity of the Object***. Recall that all of the objects are represented as a set of surfacing triangles. The more triangles used, the more complex the object will be. Also, in reality, it's more common to have objects with arbitrary shapes.
b. ***Number of Objects***. The more objects involved, the more object pairs need to be checked, and the higher the number of collisions will be detected.

According to the above criterias, we designed the following cases:

- Object number = 1024, 2048, 4096
- Triangle number = 8, 24

CUDA Hyperparameter



To verify the hypothesis that the performance should not depend on the hyperparameter (Block size) selection in the CUDA program, we measure the time with different input sets, and get the result as shown in the above figure. We can see that the computation time is not changing with variant block sizes. Thus, we can conclude that the program is not going to be optimized by fine tuning the hyperparameters.

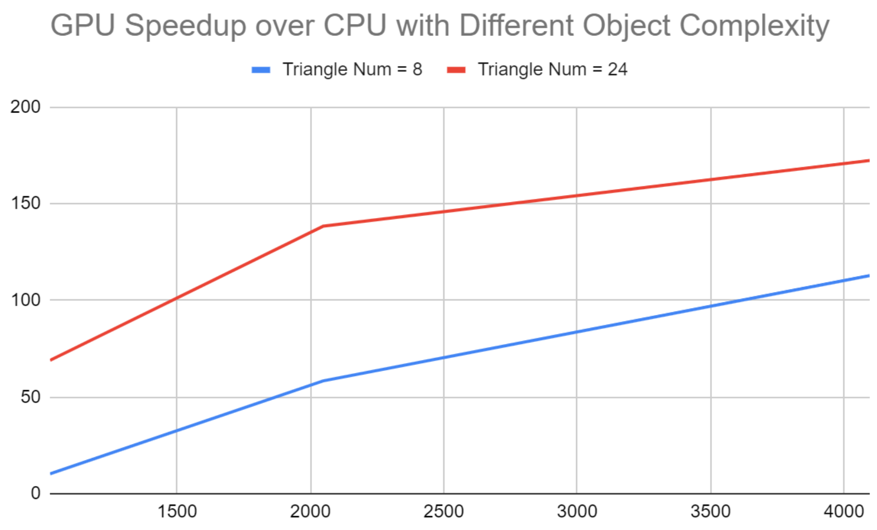
Benchmark

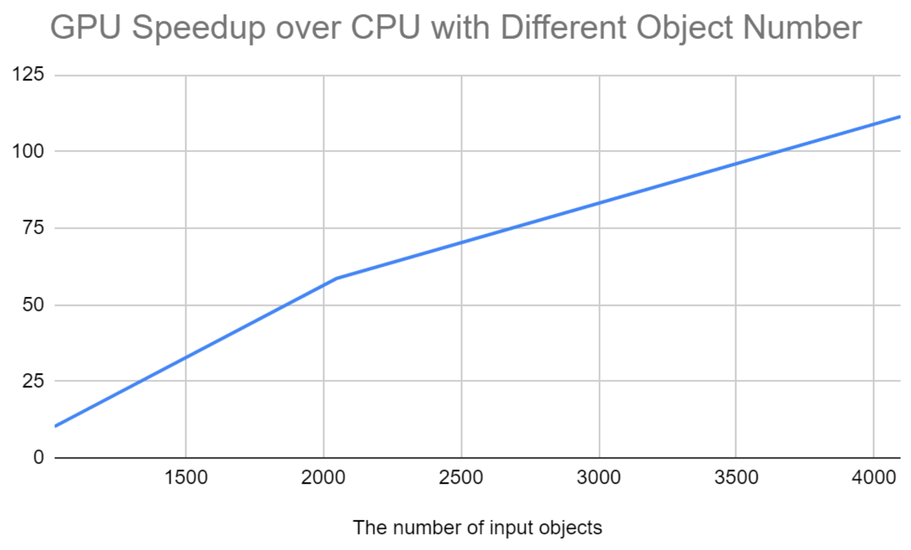
The benchmark of the following evaluation is set as the CPU version of the VCollide program. Since there are several modifications over data structure as described in the previous sections, it's not exactly the serial version of the parallelized final output.

Speedup

The speedup rate is calculated as {CPU Time} / {GPU Time}.

Triangle Number	8	8	8	24	24	24
Object Number	1024	2048	4096	1024	2048	4096
cpu time	11.3678	109.70999	381.78093	77.91455	248.25121	562.34216
gpu time	1.105	1.87986	3.38266	1.12904	1.79328	3.2619
speed up rate	10.28760181	58.36072367	112.8641158	69.00955679	138.4341598	172.3971182





Conclusion

According to the above graphs, we find that the benefit of using the GPU program is more obvious as the complexity of the input increases. We draw a conclusion that the program that we developed is achieving the goal we set at the beginning, and the mechanism selection (using CUDA) is sound.

Further Analysis

a. Workload Imbalancing & Synchronization

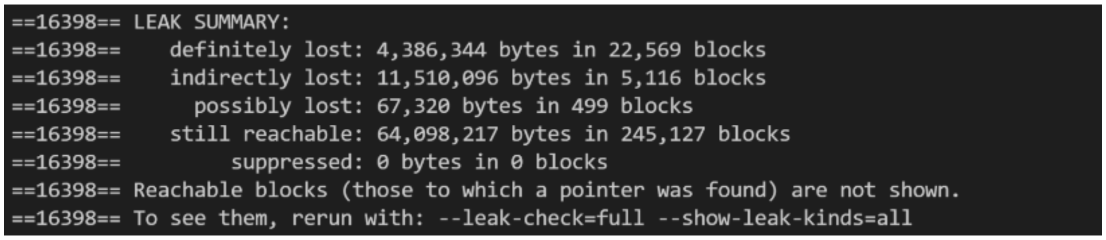
When we parallelize over a detailed collision phase, some detection is ended faster than the others. This is because a non-collision is detected at an earlier phase of the hierarchy than the others. This also negatively affects the synchronization time cost for the thread with less computation workload to wait for the incomplete threads to finish. Since this logic is inside the CUDA device code, the printing process doesn't work. Thus, we are not providing data to support the idea, but the reasoning is rational.

b. Memory Bounded Program

Our program doesn't include large amounts of data transfer. The data is mostly transferred just for the passing between the host memory space to the CUDA memory space, and everything is conducted in the CUDA logic.

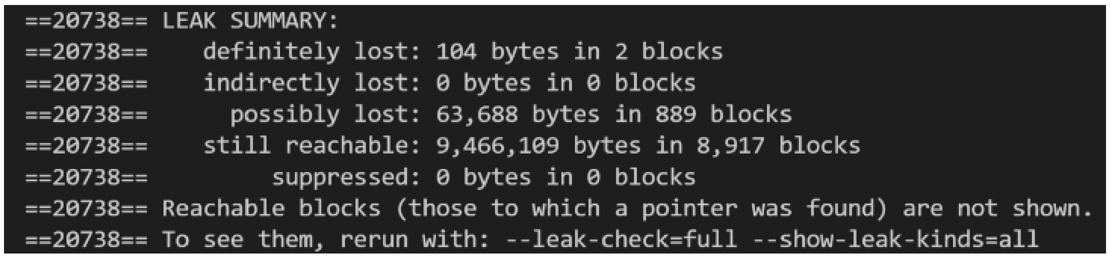
As stated in the previous section, however, the memory of CUDA space is essential for the performance and robustness of the program. With large sets of input objects and triangles, the memory occupied will significantly grow. Thus, utilizing the memory space is extremely important to help us increase the upper limit of buffer size without running out of CUDA memory.

Since the memory buffer space assignment and recycling may happen in different phases of the object's life cycle, it's not as intuitive as the "where to new, where to delete" principle. Thus, we use the tool *valgrind* to measure the capacity and pinpoint possible memory leakages. Before the memory leakage checking, our program almost ran out of the CUDA heap according to the valgrind report – matching with our prediction. Thus, we add the pairing of memory buffer freeing instructions accordingly.



```
==16398== LEAK SUMMARY:
==16398==    definitely lost: 4,386,344 bytes in 22,569 blocks
==16398==    indirectly lost: 11,510,096 bytes in 5,116 blocks
==16398==      possibly lost: 67,320 bytes in 499 blocks
==16398==    still reachable: 64,098,217 bytes in 245,127 blocks
==16398==         suppressed: 0 bytes in 0 blocks
==16398== Reachable blocks (those to which a pointer was found) are not shown.
==16398== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```

Memory Leakage Before Optimizing



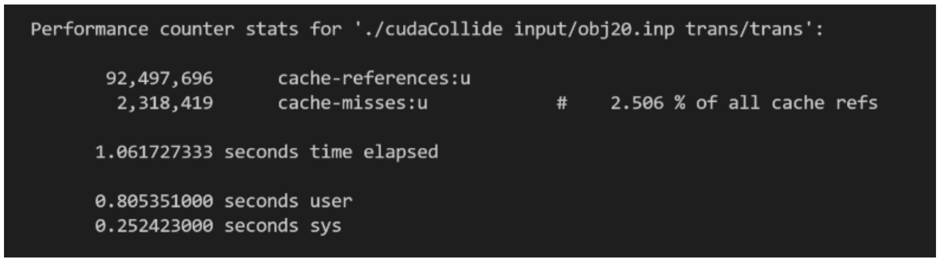
```
==20738== LEAK SUMMARY:
==20738==    definitely lost: 104 bytes in 2 blocks
==20738==    indirectly lost: 0 bytes in 0 blocks
==20738==      possibly lost: 63,688 bytes in 889 blocks
==20738==    still reachable: 9,466,109 bytes in 8,917 blocks
==20738==         suppressed: 0 bytes in 0 blocks
==20738== Reachable blocks (those to which a pointer was found) are not shown.
==20738== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```

Memory Leakage After Optimizing

It’s noteworthy that, in comparison with the data structure of linked-list, the repeated buffer allocation-free instructions has a negative effect over the speed of the whole program. However, the robustness of the program depends on the paired memory interaction instructions. If the memory space is never released, when the number of triangles-per-object or the number of objects or the frame number increases, the program can easily crash.

c. Cache Miss Rate

We have checked the cache performance using the tool *perf*. According to the below screenshot, the cache miss rate is 2.506%. Thus, we don’t regard rearranging the memory accessing method as an important target during the optimization.



```
Performance counter stats for './cudaCollide input/obj20.inp trans/trans':

       92,497,696      cache-references:u
        2,318,419      cache-misses:u           #    2.506 % of all cache refs

      1.061727333 seconds time elapsed

      0.805351000 seconds user
      0.252423000 seconds sys
```

Deep Analysis

We can divide our work into 2 phases: broad phase and narrow phase. We analyzed the running time for both the relative slow movement case and relative fast movement phase. Below is the running time for 1 simulation step using 1024 objects in seconds.

	Broad	Narrow
Slow movement	0.00194	0.01075
Fast movement	0.10240	0.01019

We can see that the time spent on the broad phase varies largely for different movements but stays stable for the narrow phase. This is primarily due to the variation of sorting as discussed in the last section. More stable sorting algorithm can be used to improve results. On the other hand, to run the recursion locally without stack and queues, I need to convert the code from dfs algorithm to bfs algorithm. Finding a way to do the dfs locally might further optimize the performance.

REFERENCES

- Starter Code: V-Collide Algorithm
- Cuda Collision Detection

LIST OF WORK

The distribution lies as 50%-50% for Hao and Tianxiang.

Hao Ding

- Conduct background research.
- Select and configure the starter code according to our project goal.
- Migrate and parallelize the code with CUDA.
- Validate and measure the performance of the parallelized version.

Tianxiang Gao

- Conduct background research.
- Select and configure the starter code according to our project goal.
- Customize input set for the project.
- Simulate the output over matlab.
- Validate and measure the performance of the parallelized version.