Noname manuscript No.
(will be inserted by the editor)

Comparative Investigation of GPU-Accelerated Triangle-Triangle Intersection Algorithms for Collision Detection

Lei Xiao · Gang Mei · Salvatore Cuomo · Nengxiong Xu

Received: date / Accepted: date

Abstract Efficient collision detection is critical in 3D geometric modeling. In this paper, we first implement three parallel triangle-triangle intersection algorithms on a GPU and then compare the computational efficiency of these three GPU-accelerated parallel triangle-triangle intersection algorithms in an application that detects collisions between triangulated models. The presented GPU-based parallel collision detection method for triangulated models has two stages: first, we propose a straightforward and efficient parallel approach to reduce the number of potentially intersecting triangle pairs based on AABBs, and second, we conduct intersection tests with the remaining triangle pairs in parallel based on three triangle-triangle intersection algorithms, i.e., the Möller's algorithm, Devillers' and Guigue's algorithm, and Shen's algorithm. To evaluate the performance of the presented GPU-based parallel collision detection method for triangulated models, we conduct four groups of benchmarks. The experimental results show the following: (1) the time required to detect collisions for the triangulated model consisting of approximately 1.5 billion triangle pairs is less than 0.5 s; (2) the GPU-based parallel collision detection method speedup over the corresponding serial version is 50x - 60x, and (3) Devillers' and Guigue's algorithm is comparatively and comprehensively the best of the three GPU-based parallel triangle-triangle intersection algorithms. The presented GPU-accelerated method is capable of efficiently detecting the potential collisions of triangulated models. Overall, the GPU-accelerated parallel Devillers' and Guigue's triangle-triangle intersection algorithm is recommended when performing practical collision detections between large triangulated models.

Keywords Collision Detection · Triangle-Triangle Intersection · Graphics Processing Unit · Parallel Algorithm

L. Xiao · G. Mei · N. Xu
School of Engineering and Technolgy,
China University of Geosciences, Beijing, China
E-mail: gang.mei@cugb.edu.cn
xunengxiong@cugb.edu.cn

S. Cuomo
Department of Mathematics and Applications ”R. Caccioppoli”,
University of Naples Federico II, Italy
E-mail: salvatore.cuomo@unina.it

1 Introduction

With the rapid development of new generation information technologies such as the IoT [1,2], multimedia data has become increasingly important as a research object for information analysis. As one type of multimedia data, 3D geometric models play a basic yet critical role in various science and engineering applications. Efficient collision detection is critical in 3D geometrical modeling; however, with the continuous refinement and scale expansion of 3D models (i.e., with a large number of elements), current collision detection methods are also computationally inefficient, making this problem an important scientific challenge.

In general, 3D geometric models are considered and interpreted as polygonal objects, such as triangulated models [3,4]. Collision detection of the 3D geometric models is then transformed into detecting the potential collision of these triangulated models (i.e., the detection of potential intersections among a large number of triangulated pairs).

Collision detection of triangulated models typically involves two stages: locating potentially intersected pairs of primitives (e.g., triangles) and determining whether each pair of primitives intersects. Therefore, there are two common strategies for improving the effectiveness and efficiency of collision detection for triangulated models. The first and most popular strategy is to reduce the number of potentially intersected primitives using (1) various BVHs, such as an AABB

[5], sphere, oriented bounding box (OBB) [3,6,7], DOPs or convex hull, and (2) various spatial decomposition structures, such as a uniform grid [8,9] or and octree grid [10]. The second strategy is to design faster algorithms for detecting the potential intersections of many pairs of primitives. Adopting high-performance architectures or methods such as multicore CPUs [11,12], many-core GPUs [13–15], FPGA [16], or even cloud computing [17,18] can be helpful when using this strategy.

Much research work has been conducted to reduce the number of potentially intersected primitives using various BVHs and spatial decomposition structures, such as a grid or various trees. For example, Vigueras et al. [19] accelerated collision detection for large-scale crowd simulations on multicore and many-core architectures based on a grid. Wang et al. [20] presented a GPU-based fast and robust BVH-based collision detection scheme specifically for deformable objects by ordering and restructuring BVHs. Fan et al. [21] proposed a new octree-based proxy for particles colliding with meshes on the GPU. Wong et al. [10] employed an adaptive octree grid for GPU-based collision detection of deformable objects.

In addition, considerable research has been conducted to design and develop parallel algorithms to rapidly detect potential intersections among pairs of primitives on heterogeneous architectures. By combining a multicore CPU and a many-core GPU, Mazhar et al. [22] developed an efficient parallel algorithm to detect collisions between complex bodies that represent large collections of spheres. Similarly, Vigueras et al. [19] accelerated collision detection for large-scale crowd simulations using multicore and many-core architectures based on a grid. Hendrich et al. [23] presented a BVH construction algorithm for a multicore CPU by using an auxiliary BVH that significantly decreased the workload.

Thompson et al. [24] developed a parallel GPU implementation for conflict detection and applied it to aircraft trajectory planning. Pan and Manocha [25] presented a parallel clustering scheme and collision-packet traversal to perform efficient collision queries on a GPU for sample-based motion planning. Laccetti et al. [26] proposed a new coordinated approach to manage heap-based priority queues that effectively improved the performance of these dynamical data structures. Weller et al. [4] defined a novel geometric predicate and a class of objects that enables us to prove a linear bound on the number of intersecting polygon pairs for colliding 3D objects on the GPU. To improve the intersection testing speed in the ray tracing algorithm, Zheng [27] proposed a fast GPU-accelerated parallel algorithm of triangle-triangle intersection by octree subdivision. Ye et al. [28] optimized the stability of the intersection algorithm in projection classification, singular case processing and exclusion separation. The GPU implementation was approximately 13 times faster than the original algorithm.

In this study, we implemented three parallel triangle-triangle intersection algorithms on GPUs to perform collision detections of triangulated models. Our goal is to first compare the computational efficiency of the three GPU- accelerated parallel triangle-triangle intersection algorithms for detecting potential intersections of triangulated models consisting of a large number of triangle primitives and then to make recommendations.

There are two stages in the presented GPU-based parallel collision detection methods for triangulated models. First, we propose a straightforward and efficient parallel approach to reduce the number of potentially intersected triangle pairs based on AABBs. Second, we develop GPU-based parallel algorithms to test the intersections of the remaining triangle pairs based on the three triangle-triangle intersection algorithms.

A key procedure in the design of fast parallel algorithms for detecting collisions of complex triangulated models is to develop an efficient triangle-triangle intersection algorithm. This can be realized by utilizing the massively parallel computing capabilities of modern GPUs. Several famous algorithms for calculating triangle-triangle intersections exist [29], including Möller's algorithm [30], Held's algorithm [31], Devillers' and Guigue's algorithm [32,33], Shen's algorithm [34], and Tropp's algorithm [35]. In this paper, we employ three commonly used algorithms (i.e., Möller's, Devillers' and Guigue's, and Shen's algorithm).

Specifically, Möller's algorithm is the most popular algorithm. It simplifies a triangle intersection into the intersection of two-line segments or the intersection of segments and triangles, which is reference for other algorithms. Devillers' and Guigue's algorithm uses a vector to determine the triangle position, while a scalar discriminant algorithm such as Möller's obtains the intersection of two triangles through accurate calculations. The differences between the implementations of these two algorithms on GPUs is worthy of research. Shen's algorithm is a vector discriminant algorithm that improves Möller's algorithm by proposing a new method to determine the positional relationship between two triangles based on the separation plane.

To evaluate the performances of the presented GPU-based parallel collision detection methods for triangulated models, we perform tests on four groups of benchmarks and investigate the computational efficiency of both the serial and parallel versions of the collision detection methods. We also compare the computational performances of the three adopted triangle-triangle intersection algorithms.

The main contributions of this paper can be summarized as follows:

(1) We implement three parallel triangle-triangle intersection algorithms on a GPU.

(2) We compare the computational efficiency of the three designed GPU-accelerated parallel triangle-triangle intersection algorithms and recommend the comparatively best parallel algorithm for collision detections of large triangulated models.

The remainder of this paper is organized as follows. Section 2 presents the background and provides an introduction to the commonly used triangle-triangle intersection algorithms. Section 3 describes the considerations and strategies for designing efficient collision detection algorithms on a GPU. Section 4 describes the four groups of benchmark tests. Section 5 discusses the experimental results. Finally, Section 6 draws several conclusions.

2 Preliminaries

In this section, we briefly introduce and compare three triangle-triangle intersection algorithms commonly used in collision detection applications: Möller's algorithm [30], Devillers' and Guigue's algorithm [32,33], and Shen's algorithm [34].

2.1 Möller's algorithm

Möller's algorithm [30] is perhaps the most commonly used approach for calculating triangle-triangle intersections in three dimensions. The essential ideas behind Möller's algorithm are straightforward. Suppose we have two triangles T_1 and T_2 in three dimensions that lie on two planes, π_1 and π_2, respectively. There are three cases for the relative location of the two planes π_1 and π_2: (1) when π_1 and π_2 are coplanar, triangle intersections in the three dimensions are reduced to special cases in two dimensions, and these special cases must be handled individually; (2) when π_1 and π_2 are parallel, the two triangles do not intersect; and (3) when π_1 and π_2 intersect.

For the third case, i.e., the two planes π_1 and π_2 intersect, first, the intersection line L of the two planes π_1 and π_2 are calculated; then, the intersection segments of the two triangles T_1 and T_2 with line L are calculated if they exist, which are noted as ij and kl, respectively. Third, if the intersection segments ij and kl exist, then the overlapped interval of the two intersection segments are checked; if the overlapped interval exists, such as segment kj in Figure 1, then the two triangles T_1 and T_2 intersect; otherwise, not.

2.2 Devillers and Guigue's algorithm

Devillers and Guigue [32] proposed a new approach to test the intersection of a pair of triangles in three dimensions. The essential idea behind this algorithm is that it was the first to define a determinant consisting of four points in three dimensions and then to determine whether two triangles intersect based on the signs of the determinants.

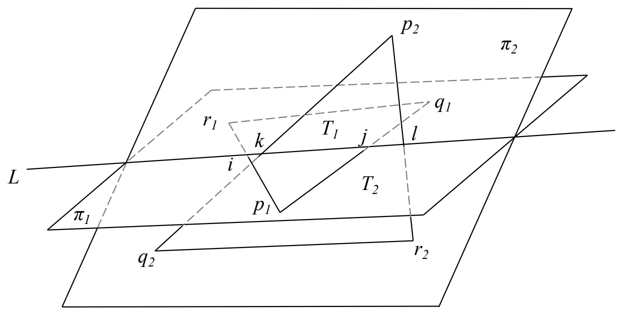


Fig. 1: Triangle-triangle intersection test in three dimensions [32]

The determinants [a, b, c, d], which consist of four points, a, b, c, and d, in three dimensions, are defined as follows,

$$[a,b,c,d] = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} = \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix},$$

where x, y and z are the coordinates.

Suppose the three points a, b and c are not collinear and form a triangle T counterclockwise, and triangle T is on the supporting plane π; if the sign of the determinant [a, b, c, d] is positive, then point d is located above the plane π; a negative determinant means the point is below the plane; and a zero means the four points are coplanar.

The algorithm proposed by Devillers and Guigue is quite like Möller's algorithm [30]. As illustrated in Figure 1, suppose there are two triangles, T_1 and T_2, that are formed by the points p_1, q_1, r_1, and p_2, q_2, r_2, respectively, and on two supporting planes, π_1 and π_2, respectively.

Devillers and Guigue's algorithm also begins to check the relative position of one triangle with the plane of the other by calculating the sign of three determinants [p_2, q_2, r_2, p_1], [p_2, q_2, r_2, q_1] and [p_2, q_2, r_2, r_1]. Furthermore, there are four cases: a) all three determinants have the same sign, and none of them are zero; b) all three determinants are zero; c) one or two determinants are zero; and d) the determinants have different signs, for example, one positive and two negatives or one negative and two positives.

Case (a) directly indicates that the two triangles do not intersect. Case (b) means the two triangles are coplanar, and this case needs to be specially handled in two dimensions. For Case (c), a point or an edge of a triangle is on the supporting plane of the other triangle; therefore, whether the point or the edge is located inside or intersects the other triangle needs to checked further. Case (d) is the most common and needs to be specially dealt with in further steps using the following predicates:

If $[p_1, q_1, p_2, q_2] \leq 0$ and $[p_1, q_1, r_2, p_2] \leq 0$, then triangles T_1 and T_2 intersect.

More detailed explanations and proof of the above predicates are described by Devillers and Guigue in the references [32,33].

2.3 Shen's algorithm

Shen's algorithm [34] to test triangle-triangle intersection is quite similar to Devillers' and Guigue's algorithm [32,33]. Based on the measurement type, the algorithm also checks whether two triangles in three dimensions intersect. The difference between the two algorithms is that Devillers' and Guigue's algorithm determines whether two triangles intersect based on the signs of the determinants, while Shen's algorithm conducts the intersection test based on the signs of the distances between line segments. More details about Shen's algorithm are presented in [34].

3 Methods

3.1 Overview

In general, 3D geometric models are represented by triangulated surface models. The detection of a potential collision, in fact, involves checking whether the triangulated surface models potentially intersect. In this paper, we focus on comparing the computational efficiency of the proposed GPU-accelerated parallel intersection algorithms of triangulated models on a CPU-GPU heterogeneous computing platform.

The presented GPU-accelerated collision detection for triangulated models involve two stages: (1) reduction of the number of potentially intersected triangle pairs based on bounding boxes and (2) further determination of the intersection test of the remaining triangle pairs using specific triangle-triangle intersection algorithms.

In this paper, a straightforward GPU-accelerated parallel algorithm for reducing the number of potentially intersected triangle pairs is proposed based on AABBs. In addition, parallel algorithms for conducting intersection tests of a large number of triangle pairs are implemented based on Möller's algorithm [30], Devillers' and Guigue's algorithm [32,33], and Shen's algorithm [34].

The flowchart of the proposed GPU-accelerated parallel algorithm is illustrated in Figure 2. More details on the two stages of the proposed parallel method are as follows.

Step 1: Calculate the AABB for the original model and the target model and denote them as `AABB1` and `AABB2`, respectively;

Step 2: Calculate the overlap between `AABB1` and `AABB2`, and denote it as `ovlAABB`;

Step 3: For each triangle primitive in the **first** model, create the AABB and then check whether it intersects with `ovlAABB`; if so, flag the triangle as 1 (i.e., `true`); otherwise, flag it as 0 (i.e., `false`). After the check is complete, remove the triangles flagged as 0.

Step 4: For each triangle primitive in the **second** model, create the AABB and then check whether it intersects with the `ovlAABB`; if so, flag it as a 1 (i.e., `true`); otherwise, flag it as 0 (i.e., `false`). After the check is complete, remove the triangles flagged as 0.

Step 5: For each of the remaining triangles in the **first** model, create the AABB and then check whether it intersects with the AABB of each of the remaining triangles in the **second** model. If so, record the IDs of the pair of potentially intersected triangles.

Step 6: Conduct the final intersection test for each pair of potentially intersected triangles using Möller's algorithm [30], Devillers' and Guigue's algorithm [32,33], or Shen's algorithm [34].

3.2 Stage 1: Reducing the Number of Potentially Intersected Triangle Pairs

One essential idea in this stage is that when predicting collisions, if the first model intersects with the second model, then the intersected part must fall into the overlap between the AABBs of the models. Consequently, we can easily filter out a large number of nonintersected triangle pairs and thus reduce the number of potentially intersected triangle pairs.

To perform the above filtering procedure to reduce the redundant triangle pairs, first, the AABBs for both the first model and the second model are calculated (denoted as `AABB1` and `AABB2`, respectively). Then, the overlap between `AABB1` and `AABB2` (denoted as `ovlAABB`) is calculated.

The AABB for each triangle primitive in both models is created and then its intersection with `ovlAABB` is checked; if it intersects, the triangle is flagged with a 1 (i.e., `true`); otherwise, it is flagged as 0 (i.e., `false`). Note that this step is quite suitable for parallelization on a GPU. Each GPU thread is invoked to check whether a specific triangle intersects the `ovlAABB`; then, the flag is marked.

After the checks are complete, the triangles flagged as 0 can be removed from the list of triangles. The removal procedure can also be parallelized on the GPU. First, all the triangles are sorted by their the flags in descending order using a GPU-based parallel sort routine. Then, two additional GPU-accelerated parallel routines, `thrust::resize()` and `thrust::shrink_to_fit()` are used to remove triangles 0 flags.

After the above first round of filtering, only a small number of triangles will remain in both models. For each of

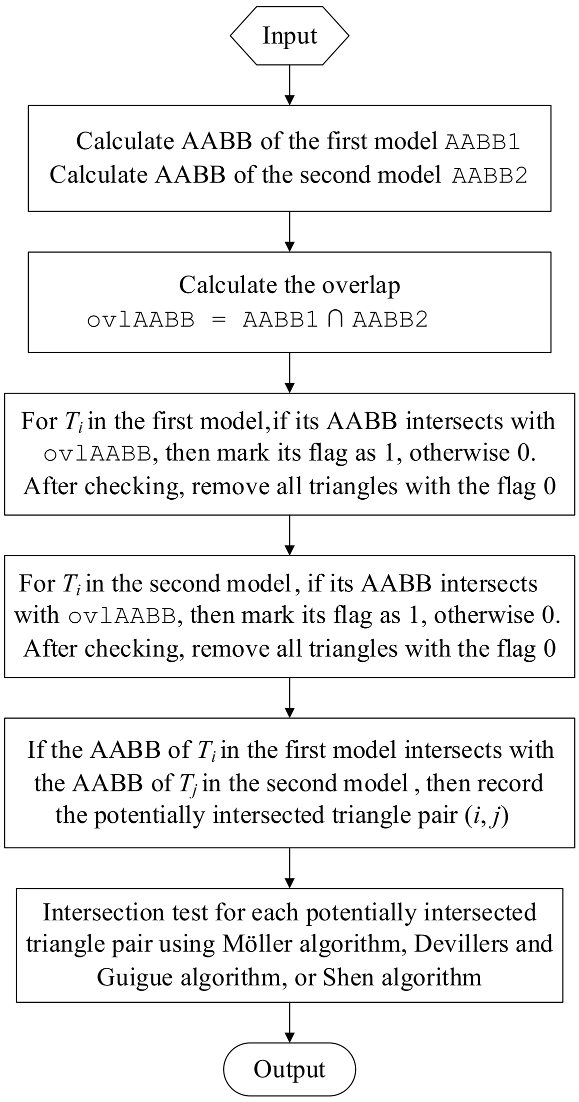

Input
Calculate AABB of the first model AABB1
Calculate AABB of the second model AABB2
Calculate the overlap
ovlAABB = AABB1 ∩ AABB2
For T_i in the first model,if its AABB intersects with ovlAABB, then mark its flag as 1, otherwise 0. After checking, remove all triangles with the flag 0
For T_i in the second model, if its AABB intersects with ovlAABB, then mark its flag as 1, otherwise 0. After checking, remove all triangles with the flag 0
If the AABB of T_i in the first model intersects with the AABB of T_j in the second model, then record the potentially intersected triangle pair (i, j)
Intersection test for each potentially intersected triangle pair using Möller algorithm, Devillers and Guigue algorithm, or Shen algorithm
Output

Fig. 2: Flowchart of the proposed GPU-based parallel collision detection method

the remaining triangles in the **first** model, we calculate its AABB and then check whether it intersects with the AABB of each of the remaining triangles in the **second** model. When a potential intersection is found, the IDs of the pair of potentially intersected triangles will be recorded. This step is also quite suitable for GPU parallelization. Each GPU thread is responsible for checking whether one pair of remaining triangles intersect.

3.3 Stage 2: Determining the Intersection Test for the Remaining Triangle Pairs

Several famous algorithms exist for testing the intersection of a pair of triangles, including Möller's algorithm [30], Devillers' and Guigue's algorithm [32], and Shen's algorithm [34]. Among these, Möller's algorithm is the most popular; however, it has been reported that both Devillers' and Guigue's algorithm [32] and Shen's algorithm [34] execute slightly faster than Möller's algorithm on the CPU. However, no works have reported a comparative analysis of the efficiency of these algorithms on a GPU.

In this paper, we employ all three of the above algorithms to develop a parallel way to determine the intersection of a pair of triangles. The parallelization process is straightforward. For each pair of triangles among the remaining triangles, an intersection test is conducted using one of the aforementioned three algorithms. Obviously, no data dependency exists between the intersection tests for any two pairs of triangles; thus, the intersection tests can be conducted in parallel. More specifically, each GPU thread is responsible for checking whether two triangles intersect. Within each GPU thread, we invoke Möller's algorithm [30], Devillers' and Guigue's algorithm [32], and Shen's algorithm [34] for comparative purposes.

3.4 GPU-based Parallelization Considerations

3.4.1 Data Layout Selection

Data layout is the form in which multivalued data such as three-dimensional points are organized and accessed in memory [36]. Selecting a proper data layout is critical in GPU-accelerated applications because the same application may display significantly different efficiencies when using different data layouts. In general, there are two main data layouts: the AoS and the SoA (see Figure 3).

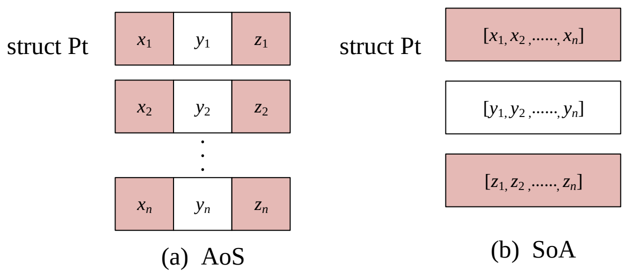


Fig. 3: Two common data layouts: AoS and SoA [36]

The use of the AoS layout may cause coalescing issues when the data are interleaved, while the SoA layout can generally make full use of the memory bandwidth because no data interleaving is involved. However, Giles et al. [37] preferred using the AoS layout with the _align_ option to store mesh data because that layout provides better memory access performance. In addition, Mei et al. [36] reported that for the interpolation of 3D points, the AoS layout always achieves better efficiency than does the SoA layout. Thus, in this work, we adopt the AoS layout.

Another notable issue when exploring the AoS layout is the use of built-in data types. CUDA provides various built-in data types (e.g, `float2`, `float4`, or `int4`). Alignment is automatically fulfilled for the above built-in data types to avoid introducing coalescing issues when the data are interleaved. Moreover, the memory bandwidth and the L2 cache memory can be fully utilized with the built-in data types. In this work, we employ the built-in `float4` and `int4` types to store the points and triangles for the developed GPU implementations.

3.4.2 Utilization of Efficient Parallel Primitives

The proposed parallel collision detection algorithm requires sorting a list of triangles and removing redundant triangles from a list. These sorting and removal procedures are computationally expensive for large numbers of triangles. To improve the computational efficiency, we utilize highly efficient parallel primitives such as the parallel sorting provided by the ***Thrust*** library [38]. GPU-accelerated parallel primitives are quite efficient and robust. Employing these parallel primitives helps to improve the efficiency and reliability of the proposed parallel collision detection algorithm implementations.

4 Results

4.1 Experimental Environment and Testing Data

4.1.1 Experimental Environment

To evaluate the performance of the proposed GPU-accelerated parallel collision detection algorithm, we conducted four groups of benchmark tests on a workstation computer, whose specifications are as follows.

Table 1: Specifications of the workstation computer for performing benchmark tests

Specifications	Details
CPU	Intel Xeon Gold 5118 CPU
CPU Frequency (GHz)	2.30
CPU RAM (GB)	128
GPU	Quadro P6000
GPU memory (GB)	24
CUDA cores	3840
OS	Windows 10 Professor
Compiler	VS2015 Community
CUDA version	v9.0

4.1.2 Testing Data

To evaluate the computational efficiency of the proposed GPU-accelerated collision detection algorithm, we generated four groups of testing data (see Figure 4 and Table 2). Each group of testing data involves two subvirtual models.

4.2 Experimental Results

For the aforementioned four groups of testing data, we evaluated the computational efficiency of both serial and parallel collision detection implementations for the three adopted triangle-triangle intersection algorithms (Möller's algorithm [30], Devillers and Guigue's algorithm [32], and Shen's algorithm [34]); see Tables 3 $\sim$ 5, respectively. Note that the computational time spent reading the files of testing data was not considered and was not included in the running times listed in Tables 3 $\sim$ 5.

Moreover, for the aforementioned four groups of testing data, we evaluated the effectiveness of the process for filtering the redundant triangle pairs. More specifically, the percentage of the triangle pairs remaining after filtering was investigated; the details are listed in Table 6. These results indicate that more than 99% of the triangle pairs are filtered, leaving only a small number of triangle pairs.

5 Discussion

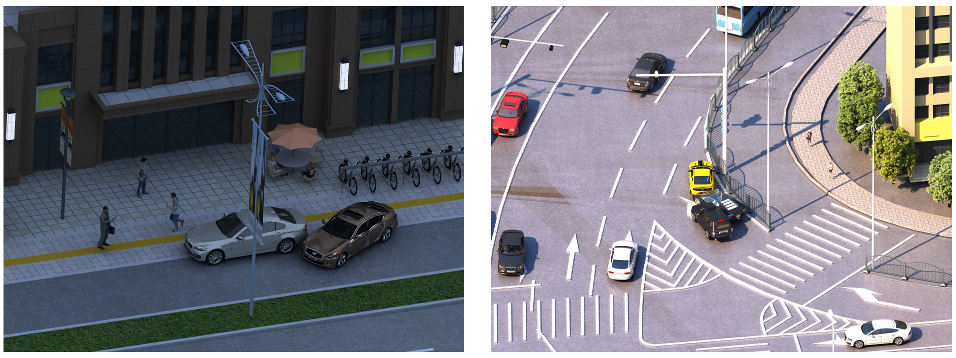
In this section, we discuss the filtering performance for redundant triangle pairs and compare the efficiency of the three adopted triangle-triangle intersection algorithms.

5.1 Performance of the Filtering of Redundant Triangle Pairs

Collision detection is a critical issue in 3D geometrical modeling. The computational time for collision detection between complex and large geometrical models composed of numerous triangles needs to be optimized. To improve the computational efficiency of collision detection, we filter the redundant triangle-triangle intersection tests using an AABB-based algorithm. Zheng [27] used the octree spatial subdivision algorithm for the same purpose. The octree algorithm divides the target scene into eight subgrids, allowing the triangle pairs located in different grids to be filtered rapidly. The octree algorithm eliminates redundant triangle pairs before determining the target model; therefore, it has a wider application scope. However, this more general algorithm also requires lengthier execution times. In addition, when the number of triangles in a subbounding box exceeds a threshold, that subbounding box will be further subdivided,

Table 2: Four groups of testing data

Test data	Number of triangles of the first model	Number of triangles of the second model	Illustration
1	14828	114489	Figure 4(a)
2	13371	94074	Figure 4(b)
3	23385	63784	Figure 4(c)
4	14621	92381	Figure 4(d)


(a) (b)

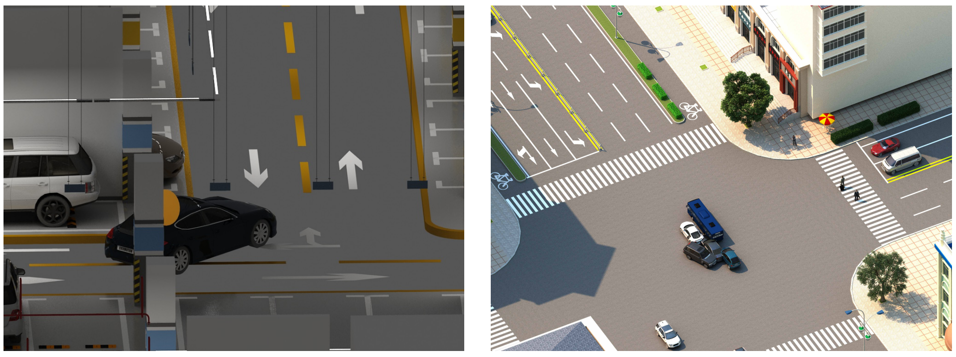

(c) (d)

Fig. 4: Rendered scenes of the four employed testing datasets

Table 3: Computational efficiency of the serial and parallel versions of the collision detection method when using Möller's algorithm

Test data	Serial Version (On the CPU)			Parallel Version (On the GPU)			Speedup
	Stage 1	Stage 2	Total	Stage 1	Stage 2	Total	
1	2061.78	1525.75	3587.53	49.5	2.1	51.6	69.53
2	3918.13	12535.4	16453.53	262.5	19	281.5	58.45
3	3383.78	7135.36	10519.14	176.3	7.8	184.1	57.14
4	6654.62	20092.4	26747.02	442.8	29.7	472.5	56.61

which makes the partitioning process data dependent and has a substantial impact on the algorithm's parallel efficiency. Therefore, while the octree algorithm has a wider application range, its theoretical speed is relatively slow. The filtering algorithm is straightforward, andmost importantlyit is highly suitable for GPU parallelization. The results indicate that more than 99% of the triangle pairs are filtered, leaving only a small number of triangle pairs to be processed. This result indicates that the filtering algorithm is effective. However, compared to the total computational time consumed by the entire collision detection procedure, the computational time of the filtering procedure is still too high.

To identify this computational efficiency bottleneck, we recorded the computational time for each critical piece of the GPU code, including GPU memory allocation, the GPU kernel invocation, and array sorting (i.e., sorting the list of triangles), and observed that allocating a large array to record the indices for the triangles of potential interest is the most computationally expensive operation because it requires a large amount of GPU memory. One possible solution is to filter

Table 4: Computational efficiency of the serial and parallel versions of the collision detection method when using Devillers and Guigue's algorithm

Test data	Serial Version (On the CPU)			Parallel Version (On the GPU)			Speedup
	Stage 1	Stage 2	Total	Stage 1	Stage 2	Total	
1	1842.22	1440.68	3282.9	45.7	2.1	47.8	68.68
2	3934.79	12314.4	16249.19	258.5	18.2	276.7	58.72
3	3152.51	6124.3	9276.81	174.4	7.9	182.3	50.89
4	6626.02	17402.6	24028.62	440.3	28.1	468.4	51.30

Table 5: Computational efficiency of the serial and parallel versions of the collision detection method when using Shen's algorithm

Test data	Serial Version (On the CPU)			Parallel Version (On the GPU)			Speedup
	Stage 1	Stage 2	Total	Stage 1	Stage 2	Total	
1	1773.42	1429.51	3202.93	50.2	2.3	52.5	61.01
2	3914.86	12921.1	16835.96	285.1	21.3	306.4	54.95
3	3282.88	6459.42	9742.3	191.8	8.1	199.9	48.74
4	6971.75	19805.8	26777.55	477.4	31.7	509.1	52.60

Table 6: Number and percentage of remaining triangle pairs after filtering

Test data	Number of triangles of the first model	Number of triangles of the second model	Number of remaining triangle pairs after filtering	Percentage of remaining triangle pairs after filtering
1	14828	114489	446566	0.026%
2	13371	94074	3798127	0.302%
3	23385	63784	746302	0.050%
4	14621	92381	5241726	0.338%

out many more triangle pairs that cannot possibly be intersected and retain fewer possibly intersected triangle pairs by using a more accurate filter procedure. Therefore, in future work, we plan to design a more efficient filtering algorithm that is highly suitable for parallelization on the GPU.

5.2 Comparative Analysis of Three Triangle-triangle Intersection Algorithms

In the presented GPU-accelerated collision detection algorithm, after filtering the redundant triangle pairs (i.e., stage 1-removing nonintersected triangle pairs), we apply three triangle-triangle intersection algorithms to further calculate the intersections of the remaining triangle pairs (i.e., stage 2-conducting accurate triangle pair intersection tests). We compare the computational efficiency of both the serial and parallel versions of the three triangle-triangle intersection algorithms (see Figures 5 $\sim$ 6).

We observed that among the serial versions, Devillers and Guigue's algorithm is slightly faster than both Möller's algorithm and Shen's algorithm. This result occurs because Devillers and Guigue's algorithm involves fewer arithmetic operations than does of Möller's algorithm.

Regarding the parallel versions developed for the GPU, Devillers and Guigue's algorithm is still slightly faster than both Möller's algorithm and Shen's algorithm, but its performance gain is less significant; conducting massive parallel computing operations on the GPU reduces the performance improvement of Devillers and Guigue's algorithm over Möller's algorithm and Shen's algorithm.

The serial Shen's algorithm achieves approximately the same computational efficiency as does Möller's algorithm; however, the parallel Shen's algorithm is the slowest of the GPU implementations, which indicates that Shen's algorithm is not as suitable for GPU parallelization. This conclusion can also be verified based on the speedups achieved by the parallel version of Shen's algorithm compared to the serial version (see Figure 6).

Möller's algorithm [30] is an all-arithmetic method that is influenced by computer errors and accumulative errors, while Devillers and Guigue's algorithm [32] uses geometric predicates to conduct the intersection test. As a result, Devillers and Guigue's algorithm is highly efficient and requires minimum arithmetic precision; thus, it has strong robustness and stability. Therefore, after comprehensive consideration, we suggest employing Devillers and Guigue's algorithm when developing GPU-based parallel collision detection applications.

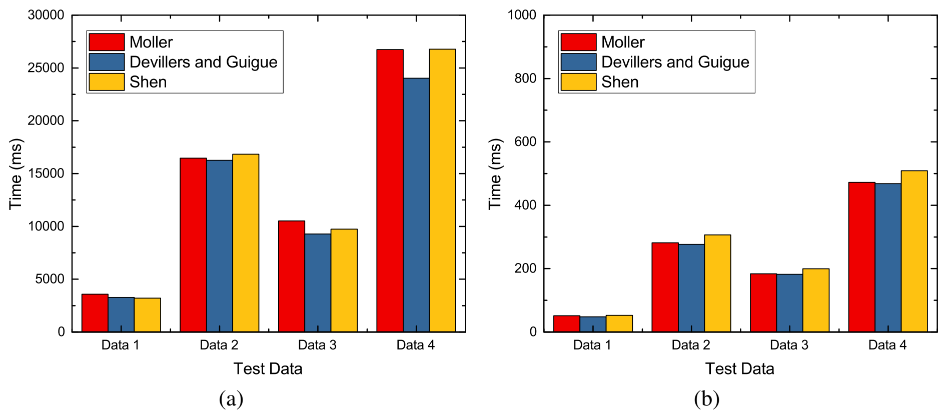


Fig. 5: Comparison of the computational time of collision detection when using three triangle-triangle intersection algorithms

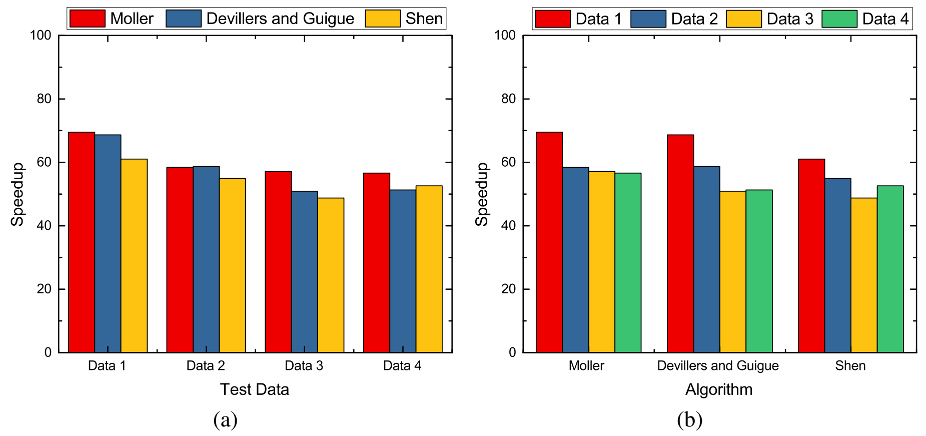


Fig. 6: Comparison of the speedups of the parallel version over the serial version of collision detection when using three triangle-triangle intersection algorithmsComparison of the speedups of the parallel version over the serial version of collision detection when using three triangle-triangle intersection algorithms

5.3 Limitation

The first shortcoming of the parallel implementation is that the efficiency of the triangle intersection algorithm is affected by the test scene to some extent. When the triangles of the testing models are too scattered or discontinuous, most triangle pairs will be excluded in advance, which is not conducive to full exploitation of the power of parallel algorithms.

Another limitation is that triangle intersection algorithms must judge the relative position among the midpoints, lines and faces of triangles. Therefore, the code contains many conditional control statements; these have a large impact on the performance of GPUs, which have weak branch prediction abilities.

6 Conclusion

In this paper, we implemented three parallel triangle-triangle intersection algorithms for collision detection on a GPU. We also compared the computational efficiency of three GPU-accelerated parallel triangle-triangle intersection algorithms for collision detection of large triangulated models. To evaluate the performance of the three presented GPU-accelerated collision detection methods, we conducted experiments with four groups of benchmarks. The results showed the following: (1) performing collision detection for approximately 1.5 billion triangle pairs requires less than 0.5 s; (2) the GPU-based parallel collision detection methods provide speedups over the corresponding serial version of 50x - 60x, and (3) Devillers and Guigue's algorithm outperforms the other two algorithms tested in these experiments. Therefore, we recommend using the GPU-accelerated parallel Devillers and Guigue's triangle-triangle intersection algorithm for practical collision detections between large triangulated models.

Acknowledgements This research was jointly supported by the National Natural Science Foundation of China (Grant Numbers: 11602235 and 41772326), and the Fundamental Research Funds for China Central Universities (Grant Numbers: 2652018091, 2652018107 and 2652-018109). The authors would like to thank the editor and the reviewers for their contributions.

List of Abbreviations

AABB	Axially Aligned Bounding Box
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DOP	Discrete Orientation Polytope
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
IoT	Internet of Things
OBB	Oriented Bounding Box

References

1. Francesco Piccialli, Salvatore Cuomo, Vincenzo Cola, and Giampaolo Casolla. A machine learning approach for iot cultural data. *Journal of Ambient Intelligence and Humanized Computing*, 09 2019.
2. G. Mei, N. Xu, J. Qin, B. Wang, and P. Qi. A Survey of Internet of Things (IoT) for Geo-hazards Prevention: Applications, Technologies, and Challenges. *IEEE Internet of Things Journal*, pages 1–16, 2019.
3. Y. Lee and Y.J. Kim. Simple and parallel proximity algorithms for general polygonal models. *Computer Animation and Virtual Worlds*, 21(3-4):365–374, 2010.
4. R. Weller, N. Debowski, and G. Zachmann. kDet: Parallel constant time collision detection for polygonal objects. *Computer Graphics Forum*, 36(2):131–141, 2017.
5. X. Zhang and Y.J. Kim. Interactive collision detection for deformable models using streaming AABBs. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):318–329, 2007.
6. Rong Zhang, Xiaogang Liu, and Jianjun Wei. Collision detection based on OBB simplified modeling. *Journal of Physics: Conference Series*, 1213:042079, jun 2019.
7. W. Zhao, C.-S. Chen, and L.-J. Li. Parallel collision detection algorithm based on OBB tree and MapReduce. *Lecture Notes in Computer Science*, 6249:610–620, 2010.
8. W. Fan, B. Wang, J.-C. Paul, and J. Sun. A hierarchical grid based framework for fast collision detection. *Computer Graphics Forum*, 30(5):1451–1459, 2011.
9. B. Yong, J. Shen, H. Sun, H. Chen, and Q. Zhou. Parallel GPU-based collision detection of irregular vessel wall for massive particles. *Cluster Computing*, 20(3):2591–2603, 2017.
10. T.H. Wong, G. Leach, and F. Zambetta. An adaptive octree grid for GPU-based collision detection of deformable objects. *Visual Computer*, 30(6-8):729–738, 2014.
11. S. Pei, M.-S. Kim, and J.-L. Gaudiot. Extending amdahl's law for heterogeneous multicore processor with consideration of the overhead of data preparation. *IEEE Embedded Systems Letters*, 8(1):26–29, 2016.
12. S. Pei, J. Zhang, N. Xiong, M.-S. Kim, and J.-L. Gaudiot. Energy efficiency of heterogeneous multicore system based on the enhanced amdahl's law. *International Journal of High Performance Computing and Networking*, 12(3):261–269, 2018.
13. X. Qi, C. Liu, and S. Schuckers. Key-frame analysis for face related video on GPU-accelerated embedded platform. pages 682–687, 2017.
14. Raffaele Montella, Giulio Giunta, Giuliano Laccetti, Marco Lapegna, Carlo Palmieri, Carmine Ferraro, Valentina Pelliccia, Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. On the virtualization of cuda based gpu remoting on arm and x86 machines in the gvirtus framework. *International Journal of Parallel Programming*, 45(5):1142–1163, Oct 2017.
15. Raffaele Montella, Diana Di Luccio, Livia Marcellino, Ardelio Galletti, Sokol Kosta, Giulio Giunta, and Ian Foster. Workflow-based automatic processing for internet of floating things crowdsourced data. *Future Generation Computer Systems*, 94:103 – 119, 2019.
16. T.M. Platt and C. Liu. Reducing test time with FPGA accelerators using OpenCL. pages 1–9, 2018.
17. Raffaele Montella, David Kelly, Wei Xiong, Alison Brizius, Joshua Elliott, Ravi Madduri, Ketan Maheshwari, Cheryl Porter, Peter Vilter, Michael Wilde, Meng Zhang, and Ian Foster. Face-it: A science gateway for food security research. *Concurrency and Computation: Practice and Experience*, 27(16):4423–4436, 2015.
18. W. Qi, J. Li, Y. Liu, and C. Liu. Planning of distributed internet data center microgrids. *IEEE Transactions on Smart Grid*, 10(1):762–771, 2019.
19. G. Vigueras, J.M. Ordua, M. Lozano, J.M. Cecilia, and J.M. Garca. Accelerating collision detection for large-scale crowd simulation on multi-core and many-core architectures. *International Journal of High Performance Computing Applications*, 28(1):33–49, 2014.
20. X. Wang, M. Tang, D. Manocha, and R. Tong. Efficient BVH-based collision detection scheme with ordering and restructuring. *Computer Graphics Forum*, 37(2):227–237, 2018.
21. W.S. Fan, B. Wang, J.-C. Paul, and J.G. Sun. An octree-based proxy for collision detection in large-scale particle systems. *Science China Information Sciences*, 56(1):1–10, 2013.
22. H. Mazhar, T. Heyn, and D. Negrut. A scalable parallel method for large collision detection problems. *Multibody System Dynamics*, 26(1):37–55, 2011.
23. Jakub Hendrich, Daniel Meister, and Jiri Bittner. Parallel bvh construction using progressive hierarchical refinement. *Computer Graphics Forum*, 36:487–494, 05 2017.
24. E. Thompson, N. Clem, D.A. Peter, J. Bryan, B.I. Peterson, and D. Holbrook. Parallel cuda implementation of conflict detection for application to airspace deconfliction. *Journal of Supercomputing*, 71(10):3787–3810, 2015.
25. Jia Pan and Dinesh Manocha. *GPU-Based Parallel Collision Detection for Real-Time Motion Planning*, pages 211–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
26. Giuliano Laccetti, Marco Lapegna, and Valeria Mele. A Loosely Coordinated Model for Heap-Based Priority Queues in Multicore Environments. *International Journal of Parallel Programming*, 44(4):901–921, Aug 2016.
27. Wang Zheng, Gaojun Ren, Liangeng Zhao, and Meijun Sun. Fast parallel algorithm of triangle intersection based on gpu. *Physics Procedia*, 33(none).
28. Xiufen Ye, Le Huang, Lin Wang, and Huiming Xing. An improved algorithm for triangle to triangle intersection test. pages 2689–2694, 08 2015.
29. G. Mei. Summary on several key techniques in 3D geological modeling. *The Scientific World Journal*, 2014, 2014.
30. Tomas Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.
31. Martin Held. ERIT - a collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.
32. Philippe Guigue and Olivier Devillers. Fast and robust triangle-triangle overlap test using orientation predicates. *Journal of Graphics Tools*, 8(1):25–32, 2003.

33. Olivier Devillers and Philippe Guigue. Faster triangle-triangle intersection tests, 2002. Report of Inria Sophia Antipolis.
34. Hao Shen, Pheng Ann Heng, and Zesheng Tang. A fast triangle-triangle overlap test using signed distances. *Journal of Graphics Tools*, 8(1):17–23, 2003.
35. Oren Tropp, Ayellet Tal, and Ilan Shimshoni. A fast triangle to triangle intersection test for collision detection. *Computer Animation and Virtual Worlds*, 17(5):527–535, 2006.
36. Gang Mei and Hong Tian. Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. *SpringerPlus*, 5(1):104, Feb 2016.
37. M.B. Giles, G.R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing OP2 for GPU architectures. *Journal of Parallel and Distributed Computing*, 73(11):1451 – 1460, 2013.
38. NVIDIA. CUDA Programming Guide. 2019.