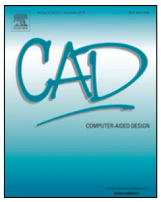
Computer-Aided Design 150 (2022) 103285

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad



Fast Parallel Evaluation of Exact Geometric Predicates on GPUs

Marcelo de Matos Menezes [a], Salles Viana Gomes de Magalhães [a,*], Matheus Aguilar de Oliveira [a], W. Randolph Franklin [b], Rodrigo Eduardo de Oliveira Bauer Chichorro [a]

[a] Universidade Federal de Viçosa, MG, Brazil
[b] Rensselaer Polytechnic Institute, Troy, NY, 12180, USA

ARTICLE INFO

Article history:
Received 31 January 2021
Received in revised form 6 November 2021
Accepted 26 April 2022

Keywords:
Geometric predicates
Parallel programming
Exact computation
Polyhedron intersection

ABSTRACT

This paper accelerates the exact evaluation of large numbers of 3D geometric predicates with an algorithm whose work is partitioned between the CPU and the GPU on a high-performance computer to exploit the relative strengths of each. The test algorithm computes all the red–blue intersections between a set of red 3D triangles and another set of blue 3D triangles. A sequence of filters is employed that progressively eliminates more and more red–blue pairs that do not intersect, finally leaving only the actual intersections. Initially, a uniform grid is constructed on the GPU to identify pairs of nearby triangles. Then, these pairs are tested for intersection with single-precision interval arithmetic on the GPU. The ambiguous cases are next filtered with double-precision interval arithmetic on the multi-core CPU, and finally the hard cases are re-evaluated in parallel on the CPU using arbitrary-precision rational numbers. The parallel speedup for the whole algorithm was up to 414 times. It took only 1.17 s to find the 18M intersections between two datasets containing a total of 14M triangles. The intersection computation was sped up by up to 1936 times. The techniques that gave this excellent performance should be useful for parallelizing other geometric algorithms in fields such as CAD, GIS, and 3D modeling.

1. Introduction

Addressing the errors caused by floating-point arithmetic is a particular challenge in computational geometry. Inexact floating-point numbers violate most of the axioms of an algebraic field, e.g., addition is not associative. Roundoff errors lead to topological errors such as causing an orientation predicate to report a point to be on the wrong side of a line segment. These errors may then propagate to higher-level operations, such as arise when using orientation predicates to compute a convex hull.

While there are heuristics, such as epsilon-tweaking and snap rounding, that try to solve this, they are not guaranteed to always work.

Representing coordinates with exact arbitrary-precision rational numbers guarantees that the computation will be free from round-off errors. However, the run-time overhead is often unacceptable. Indeed, the total number of digits in the numerator and denominator of the output is typically the sum of the numbers of digits in the operands, and so grows exponentially with the depth of the computation tree. Likewise, the time and space cost of each operation will grow exponentially with the computation tree's depth, badly degrading performance.

Some techniques have been proposed to cope. Arithmetic filters using interval arithmetic represent each exact number e as an interval $[l, h]$ of floating-point values containing e. The IEEE-754 floating-point standard guarantees that, for each arithmetic operation, a new interval guaranteed to contain the exact result can be computed. Thus, a predicate can be initially evaluated using intervals. If the exact result can be inferred from the bounds of the interval, then this result is returned. Otherwise, the expression is re-evaluated, either with exact arithmetic, or with intervals that have more precise number types. Most of the time, computation with intervals is enough to infer the exact result [1], and so predicates can be efficiently and exactly evaluated without the overhead of exact computation.

The computing capabilities of desktop computers and workstations have recently increased due to multi-core processors and accelerators such as GPGPUs (*General Purpose Graphics Processing Units*) and MICs (*Many Integrated Core Architectures*). However, many algorithms cannot take advantage of this because they are still designed for sequential architectures

This paper proposes to combine GPUs and multi-core CPUs, using the particular strengths of each, to accelerate the evaluation

* Corresponding author.
E-mail addresses: marcelo.menezes@ufv.br (M. de Matos Menezes), salles@ufv.br (S. Viana Gomes de Magalhães), matheus.a.aguilar@ufv.br (M. Aguilar de Oliveira), mail@wrfranklin.org (W. Randolph Franklin), rodrigo.chichorro@ufv.br (R.E. de Oliveira Bauer Chichorro).

https://doi.org/10.1016/j.cad.2022.103285
0010-4485/

of exact predicates using arithmetic filters. Our idea is to use the parallel processing power of the GPU to quickly evaluate a batch of predicates using interval arithmetic. GPUs are designed for fast floating point calculations, which makes them suitable for interval arithmetic. The few unreliable results are detected and re-evaluated in parallel on the CPU using exact arithmetic.

A preliminary version of this framework has been implemented and tested for computing the intersections between two sets of 2D segments [2]. This employs a uniform grid to cull the number of segments. The CPU traverses the grid and creates a list of pairs of segments, which is sent to the GPU for intersection evaluation using interval arithmetic. The results are then returned to the CPU. They are represented with a flag with 3 possible values: intersection, no intersection, or uncertain result (interval failure). Interval failure means this intersection cannot be safely determined using intervals. The CPU processes each interval failure by recomputing the intersection using multiple-precision rational numbers. Compared to a sequential implementation, the algorithm achieved speedups of up to 289 times for intersection evaluation, and up to 40 times when the entire running time is included, on an NVIDIA GTX 1070 Ti GPU.

In [3], we extended this to compute intersections between segments and triangles in 3D. The initial version performed badly because the pre-processing step, where the CPU creates a list of pairs of segments and triangles, was a bottleneck. Thus in [3] we proposed a new method using three auxiliary arrays to associate each GPU thread with the *(segment, triangle)* pairs assigned to it for testing.

This implementation led to a speedup of up to 17 times on a NVIDIA GTX 1070 Ti GPU, compared to a sequential implementation. Besides the fact that 3D predicates are more complex than 2D, which leads to more thread divergence, another reason for the smaller speedup of the 3D version is that a bounding-box culling step was added before the intersection tests. This improved the performance, but the improvement in the sequential version was better than in the parallel one. The problem was that the bounding-box test increased the GPU thread divergence. The issue is that GPU threads are grouped by 32s into *warps*, and all the threads in one warp must execute the same instruction or be idle.

Finally, in this paper, we extend the framework in three ways. First, we improve the bounding-box step to increase the parallel speedup. Second, we create the array of pairs on the GPU. Third, we use single-precision floating-point arithmetic, instead of double-precision, for the intervals, because on GPUs, single is usually several times faster. We correct for the reduced precision of single-precision as follows. Unreliable single-precision results are identified by using interval arithmetic. They are re-evaluated with double-precision, and if still unreliable, with rationals.

We evaluated these novel ideas with a new case study, described in Section 4. It detects the pairwise intersections of 3D triangles from two meshes. Experiments were performed on the newer and faster NVIDIA RTX 8000 GPU. We observed a speedup of up to 1936 times for intersection evaluation compared to a sequential implementation. The speedup for the entire running time was up to 414 times. On the older GTX 1070 Ti GPU, the parallel speedup for the intersection computation was 407×, and for the entire running time was 158×.

The performance and correctness make this technique an excellent choice for processing large datasets, where the chance of failure from inexact algorithms is higher, in interactive applications such as GIS and CAD systems.

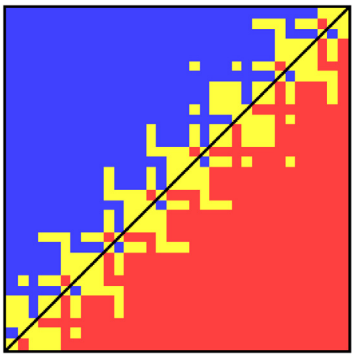


Fig. 1. Roundoff errors in the planar orientation problem — Geometry of the planar orientation predicate for double precision floating point arithmetic. Yellow, red and blue points represent, respectively, collinear, negative and positive orientations. The diagonal line is an approximation of the segment (q, r). What should happen is that all the pixels above the line are blue, and all those below are red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)
Source: [6].

2. Background

2.1. Roundoff errors

Non-integers are typically approximately represented in computers with floating-point values. The difference between a non-integer and its approximation is called the *roundoff error*. Although these differences are usually small, these errors accumulate as sequences of arithmetic operations are performed. The presence of floating point errors in computer programs can have serious consequences in diverse fields such as the failures of the first Ariane V rocket [4] and the Patriot missile defense system [5].

In geometry, roundoff errors can generate topological inconsistencies causing globally impossible results. For example, the computed intersection of two line segments may not lie on either line. Kettner et al. [6] gave some examples of failures that can cause algorithms, e.g. for computing convex hulls, to fail.

The planar orientation predicate says whether three 2D points $p = (p_x, p_y)$, $q = (q_x, q_y)$, $r = (r_x, r_y)$ are collinear, make a left turn, or make a right turn. It is the sign of the following determinant:

$$\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix}$$

Positive, negative and zero signs mean that (p, q, r), respectively, make a left turn, right turn or are collinear. Roundoff errors may cause the sign of this determinant to be evaluated wrongly, mis-classifying the orientation. To illustrate this problem, Kettner et al. [6] implemented a program to apply the planar orientation predicate ($orientation(p, q, r)$) to a point $p = (p_x + xu, p_y + yu)$, where u is the step between adjacent floating point numbers of size around p, and $0 \leq x, y \leq 255$. This results in a 256×256 matrix containing either blue, yellow and red points, where the colors mean that the corresponding point is computed to be above, on or below the line that passes through q and r. Fig. 1 shows this experiment for $p = (0.5, 0.5)$, $u = 2^{-53}$, $q = (12, 12)$ and $r = (24, 24)$. Several points have orientations computed incorrectly.

As shown by [6], these inconsistent results in the orientation predicates could make algorithms that use this predicate to fail.

There are various proposed solutions. The simplest one, *epsilon-tweaking*, uses an ϵ tolerance, and considers two values x and y equal if $|x - y| \leq \epsilon$. However this is a formal mess because

equality is neither transitive nor invariant under scaling. Thus, in practice, epsilon-tweaking can fail [6].

Snap rounding is another method to approximately represent arbitrary precision segments on a fixed-precision grid [7]. However, snap rounding can generate inconsistencies, and deform the original topology if applied repeatedly on the same dataset. Some possible solutions are presented in [8–10].

Shewchuk [11] presents the *Adaptive Precision Floating-Point* technique for exactly evaluating predicates. The evaluation is performed using the only minimum precision necessary to achieve correctness. That allows some efficient exact geometric algorithms to be developed. Many geometric predicates reduce to computing the sign of a determinant. So, the precise value of this determinant does not need to be computed as long as its sign is correct. To determine if the sign can be trusted, the approximation and an error estimate are computed. If the error is big enough to make the sign uncertain, the values are recomputed using higher precision. However, this technique is not suitable for solving all geometric problems [11]. For example, "a program that computes line intersections requires rational arithmetic; an exact numerator and exact denominator must be stored" [11].

The proper formal way to eliminate roundoff errors and guarantee algorithm robustness is to use exact computation with arbitrary precision rational numbers [6,12–14]. Computing in the algebraic field of rational numbers over the integers, with the integers allowed to grow as large as necessary, allows the traditional arithmetic operations, $+, -, \times, \div$, to be computed exactly with no roundoff error.

The cost is that the number of digits resulting from an operation is about equal to the sum of the numbers of digits in the two inputs. E.g., $\frac{214}{433} + \frac{659}{781} = \frac{452481}{338173}$. Casting out common factors helps only to a small degree. However, this growth is acceptable if the depth of the computation tree is small. Also, the cost can be significantly reduced by employing techniques such as arithmetic filtering with interval arithmetic, as we will discuss in Section 2.2.

2.2. Arithmetic filters and interval arithmetic

One technique to accelerate algorithms based on exact arithmetic is to employ arithmetic filters and interval arithmetic [15]. Each floating-point number is represented by an interval containing the exact value. During a predicate evaluation, such as the computation of the sign of an arithmetic expression, the arithmetic operations are initially applied to the intervals. The resulting interval is computed so as to guarantee that it will contain the exact result of the operation (this is called the containment property). Finally, if the bounds of the resulting interval have the same sign, so that the sign of the exact result is known, that value is returned. Otherwise, the predicate is re-evaluated using exact arithmetic instead of the floating-point intervals. The term *arithmetic filter* derives from the process of filtering the unreliable results and recomputing them with exact arithmetic.

This is possible and efficient because the IEEE-754 floating-point number standard defines how arithmetic operations are approximated: "the result of operations can be seen as if they were performed exactly, but then rounded to one of the nearest floating-point values enclosing the exact value" [15]. IEEE-754 defines three rounding-modes, selectable at runtime: rounding to the nearest representable floating-point value, or down towards $-\infty$ (i.e., the closest smaller representable floating-point number), or up towards $+\infty$ (the closest larger representable floating-point number). So, the interval containment property can be maintained.

CGAL [15] illustrates this process with addition. Suppose $xInterval = [x.lower, x.upper]$ and $yInterval = [y.lower, y.upper]$ are, respectively, floating-point intervals containing the exact values *xExact* and *yExact*. The floating-point interval $[x.lower \pm y.lower, x.upper \mp y.upper]$ (where $\pm$ and $\mp$ represent, respectively, rounding towards $-\infty$ or $+\infty$) is guaranteed to contain the exact value of the expression $xExact + yExact$.

Listing 1: Using CGAL interval arithmetic framework

```
1  // Returns true if the sum of x_exact with y_exact
2  // is positive and false otherwise.
3  // x_interval and y_interval must contain,
4  // respectively, x_exact and y_exact.
5
6  bool predicate(mpq_class x_exact,
7                 mpq_class y_exact,
8                 CGAL::Interval_nt<> x_interval,
9                 CGAL::Interval_nt<> y_interval) {
10     try {
11         if (x_interval + y_interval > 0)
12             return true;
13         else
14             return false;
15     }
16     catch (CGAL::Interval_nt<>::unsafe_comparison& ex) {
17         if (x_exact + y_exact > 0)
18             return true;
19         else
20             return false;
21     }
22 }
```

The Computational Geometry Algorithms Library (CGAL) [16] supports exact computation because its framework makes it easy to develop algorithms that combine one of its number types, arbitrary precision rational numbers, with arithmetic filters.

There are multiple types of arithmetic filters [15]. Listing 1 shows one. Here, variables with the suffix *_exact* were created as GMP [17] (*GNU Multiple Precision Arithmetic Library*) arbitrary precision rationals (of type *mpq_class*) while the ones with suffix *_interval* use CGAL's interval arithmetic number type. Both types overload the arithmetic and boolean operators. If the comparison (line 8) cannot be evaluated safely, CGAL throws an *unsafe_comparison* exception. When it is caught, that predicate can be re-evaluated using the exact version of the respective variables (line 14).

Evaluating a sequence of operations is challenging because we may not know the exact value of the operands (since they were generated by several operations). CGAL provides a more generic and reusable type of filter that solves this by using a directed acyclic graph (DAG) to represent the history of the operations that generated each geometric object.

This is transparent to the user, and does not require an explicit *try... catch* block as used in Listing 1. Another advantage is that *try... catch* blocks sometimes execute very slowly. For example, when testing *if* $(a + 2 * b + c < 0)$, intervals will be used first, to try to avoid using rationals. If the sign of $(a + 2 * b + c)$ can be evaluated, then we are done. Otherwise $(a + 2 * b + c)$ is recomputed with rationals. This is possible because the associated DAG represents the sequence of operations that computed it. This exact re-evaluation is lazily delayed until it is really needed ("as hopefully it won't be needed at all" [15]).

However these filters also have some drawbacks. The history DAG uses a lot of memory, is hard to maintain, and is not thread-safe. Even operations that do not modify the geometric objects, like the "read-only" operations in orientation predicates, often cannot execute in parallel [18].

2.3. High-performance computing and CUDA

Powerful multi-core CPUs and General Purpose GPUs (*GPGPUs*) with thousands of cores have increased the computing capability of inexpensive computers. For example, in 2020, a NVIDIA

GeForce 1070 Ti with 2432 cores, costing $700 USD, provides 8 TFLOPS of peak floating-point performance. It is important to design parallel algorithms able to use this power.

High-performance computing has been employed to accelerate some geometric algorithms. For example, the commercial product Geometric Performance Primitives (GPP) [19], performs non-exact map overlays using GPUs.

Zhou et al. [18] and Magalhães et al. [20] have developed exact parallel algorithms for shared-memory multi-core CPUs to perform boolean operations on 3D meshes. Zhou et al. [18] use CGAL routines (for example, to detect triangle–triangle intersections, to evaluate point–plane predicates, to perform Delaunay triangulations, etc.) with an exact kernel with a lazy number type. Since these operations are not thread-safe, the authors have employed mutex locks to ensure correctness. Magalhães et al. [20], on the other hand, achieved thread-safety by explicitly managing the exact arithmetic operations. For example, they implemented their own orientation predicates (using CGAL's interval arithmetic number type) and explicitly re-evaluated these predicates when the intervals were not reliable enough to ensure exactness (thus, CGALs' lazy evaluation using the history DAG was not employed in this algorithm).

While there have been exact and parallel algorithms for processing geometric data, porting these algorithms to GPUs is still a challenge, particularly when exact arithmetic operations with arbitrary-precision rationals are required. The algorithms employed in arbitrary-precision arithmetic "are not easily portable to highly parallel architectures, such as GPUs or Xeon Phi" [21]. One of the reasons for this is the typically non-trivial memory management required by this kind of computation [22].

Thus, libraries for performing higher-precision arithmetic on GPUs (such as CAMPARI [22] and GARPREC [23]) are typically designed to process extended-precision floating-point numbers.

However, thanks to arithmetic filters, floating-point operations can significantly reduce the frequency that rationals are required [1]. In this work, we combine the parallel computing capability of CPUs with GPUs for exactly performing geometric operations. The exact representation of the geometric objects is kept on the CPU, while approximate intervals (represented with floating-point numbers) are stored on the GPU. The combinatorial component of the geometric algorithms is executed on the CPU and the parallel evaluation of geometric predicates is offloaded to the GPU, which returns the exact result of each one or a flag indicating that a given predicate could not be safely evaluated with the intervals. The CPU then re-evaluates (also in parallel) the predicates that failed on the GPU.

While there has been research [24,25] on implementing interval arithmetic on GPUs, these works have focused on computer graphics applications (like ray tracing) and have not employed this technique to accelerate exact geometric computation using arithmetic filters.

3. Implementing exact parallel predicates

As stated in Section 2.2, a correct implementation of interval arithmetic relies on hardware compliance to the IEEE-754 standard. NVIDIA's GPUs' double and single precision floating point comply, starting with compute capabilities 1.3 and 2.0, respectively [26]. They adopt its newest version (IEEE-754:2008, as of June 2019), which allows the rounding criteria to be selected per machine instruction, completely removing the mode switching overhead [24].

To make the interval arithmetic transparent, we created a separate C++ class based on Collange et al. [24]. Through operator overloading, the predicate code remains clean and concise, and the compiler intrinsics are hidden from the user.

Listing 2: Some methods of our CudaInterval class

```
1
2  #define INTERVAL_FAILURE 2
3
4  class CudaInterval {
5  public:
6      __device__ __host__
7      CudaInterval(const double l, const double u)
8          : lb(l), ub(u) {}
9
10     __device__
11     CudaInterval operator+(const CudaInterval& v) const {
12         return CudaInterval(__dadd_rd(this->lb, v.lb),
13             __dadd_ru(this->ub, v.ub));
14     }
15
16     __device__
17     int sign() const {
18         if (this->lb > 0) // lb > 0 implies ub > 0
19             return 1;
20         if (this->ub < 0) // ub < 0 implies lb < 0
21             return -1;
22         if (this->lb == 0 && this->ub == 0)
23             return 0;
24
25         // If none of the above conditions is satisfied,
26         // the sign of the exact result cannot be inferred
27         // from the interval, Thus, a flag is returned
28         // to indicate an interval failure.
29
30         return INTERVAL_FAILURE;
31     }
32
33 private:
34     // Stores the interval's lower and upper bounds
35     double lb, ub;
36 };
```

For example [24], the sum of two intervals $[a, b]$ and $[c, d]$ is

$$[a, b] + [c, d] = [\underline{a + c}, \overline{b + d}]$$

where $\underline{a + c}$ and $\overline{b + d}$ indicate, respectively, the expression is rounded towards $-\infty$ and $+\infty$. Listing 2 illustrates the implementation of addition, where the CUDA C functions *__dadd_rd* and *__dadd_ru* switch the addition double precision floating point rounding mode to $-\infty$ and $+\infty$, respectively.

Our class has also the method *sign*, which returns 1, 0, or -1 if the interval's sign is guaranteed to be, respectively, positive, zero or negative. It returns a special error flag when the sign cannot be inferred from the interval's bounds. The 2D orientation predicate, described in Section 2.2, can be easily implemented on the GPU side with interval arithmetic using our class, as shown in Listing 3. However, when an interval failure occurs during the sign evaluation, the responsibility to correctly handle the case is delegated to the CPU. Nonetheless, as shown by [1], and reinforced by our case study (Sections 4 and 5) interval failures are rare and they usually do not affect the algorithms' overall performance.

GPUs are SIMT (*Single Instruction, Multiple Threads*) devices; this can be exploited by applying the same operation (for example, evaluating orientation predicates) on multiple triples of points in parallel.

Even though this example is focused on 2D orientation predicates, it can be extended to other geometric operations using interval arithmetic.

Listing 3: Orientation predicate on GPU

```
1  struct CudaIntervalVertex {
2      CudaInterval x, y;
3  };
4
5  __device__ int orientation(
```

```
6       const CudaIntervalVertex* p,
7       const CudaIntervalVertex* q,
8       const CudaIntervalVertex* r) {
9       return ((q->x - p->x) * (r->y - p->y) -
10              (q->y - p->y) * (r->x - p->x)).sign();
11  }
```

4. Fast red–blue intersection tests

We evaluated these ideas by implementing a fast exact algorithm for detecting red–blue intersections of triangles in 3D. Given T_1 red triangles and T_2 blue triangles, our objective is to find all the cases where a red triangle intersects a blue triangle. One application is to perform a quick interference cull between a red object and a blue object, each represented by a union of overlapping triangles. Most possible red–blue pairs of objects do not interfere with each other (or intersect). We wish to quickly eliminate most of the impossibilities, and then spend time on the few hard cases.

In this application, the red object will have many red–red intersections, and the blue object many blue–blue intersections, even if there are no red–blue intersections at all. That is a problem for the classic plane-sweep algorithms. They intersect the datasets with a plane and maintain a data structure describing what objects that plane intersects, and their adjacencies in that plane. This is used to compute future intersections.

As the plane sweeps up through the data, it stops at each vertex and future intersection, and updates its description to reflect adjacency changes and compute more future intersections. It is necessary to stop at all intersections, including the red–red and blue–blue ones, in order to update that description.

Even if there are no red–blue intersections, the plane sweep might need to do quadratic work. Our algorithm does not have this problem.

It executes the following series of culls, with each step reducing the number of remaining potential red–blue intersections:

- First, the triangles are indexed using a uniform grid. That is, we iterate over the triangles. For each triangle, all the grid cells that it intersects are determined, and that information is stored with each such cell.
- Then we integrate over the cells. In each cell, we iterate over all the red–blue pairs of triangles, and cull the pairs with a fast bounding-box test. The possible results of each such test are *definitely do not intersect*, or *do not know*.
- The remaining pairs are next tested with a triangle–triangle intersection algorithm using geometric predicates implemented with interval arithmetic. The possible results of each such test are *definitely intersect*, *definitely do not intersect*, or *do not know*.
- Finally, the unknown cases are re-evaluated using exact arithmetic.

Details about each step will be presented in the next subsections.

4.1. Uniform Grid Indexing

To avoid testing each triangle from T_1 against each one from T_2, which would require quadratic time in the number of triangles, we index the sets using a uniform grid. This data structure is typically employed in computational geometry to cull a combinatorial set of pairs of objects, generating a smaller subset with elements that are more likely to coincide [27]. If the input is uniformly independent and identically distributed (i.i.d.), the expected size of the resulting subset is linear on the size of the input plus the output [28–30].

This holds even though some cells might be much more populated than others, while many cells remain empty. Indeed, the number of triangles per cell is a Poisson random variable. [28–30] work this out in detail; it is also observed experimentally. A summary goes as follows.

Let $g = r^3$ be the number of grid cells and n be the number of triangle–cell incidences. An incidence is one occurrence of a triangle intersecting (or overlapping) one cell. Assume that the triangles are uniformly and independently distributed over the grid. Although there will be local exceptions to this, as $g, n \to \infty$ this is reasonable.

Let p be the probability that a given incidence occurs in a given cell. $p = 1/g$. The expected number of incidences in a given cell is n/g. Let L be the random variable for the number of incidences in any given cell. L is an example of combinatorial selection without replacement. Let l be a particular value of L. $P[L = l] = \binom{n}{l} p^l (1-p)^{n-l}$. Since $E[L] = n/g \ll n$, an excellent approximation is the Poisson distribution with parameter $\lambda = E[L]$.

If there are l triangles in a cell, the number of pairs of triangles is $\binom{l}{2} = \frac{l^2}{2} - \frac{l}{2}$. Since for a Poisson distribution, the variance is the square of the mean, the expected number of pairs of triangles in the cell, which is the time to process the cell, is on the order of the square of the mean number of incidences in the cell. With a typical choice of r, the time to process one cell is constant or slowly growing. We might pick r to keep $E[L]$ constant, or sometimes, we might let $E[l]$ grow slowly with dataset size to reduce the space. Then, the time per cell also grows slowly [28–30].

The uniform grid implementation goes as follows. Given the sets of triangles T_1 and T_2, a grid G with resolution r and containing $r \times r \times r$ cells is created covering a box that bounds T_1 and T_2 together. Next, each triangle t from either of the two input sets is inserted into all the grid cells that t's bounding-box intersects. Then we iterate over the cells c in G, testing all the pairs of red and blue triangles from each c for intersection. This intersection test is the multistep process described earlier. That finds all the pairs of intersecting triangles.

We implement the uniform grid with a ragged array, similarly to Magalhães et al. [20]. This stores a collection of arrays in a contiguous block of memory b, by using a dope vector d to keep track of each array's initial position. The jth element of the ith array is b_{j+d_i}. Since a ragged array stores the entire grid in contiguous memory [27], it is more cache friendly and uses less space than storing one dynamic array per cell. It is also easier to transfer the grid to the GPU. Finally, it parallelizes better than a dynamic array.

The ragged array is created as follows. First, the triangles are read, the cells intersecting each triangle are computed, and an array k containing the total number of triangles intersecting each cell is accumulated.

Then k is transformed into the dope vector d with a parallel prefix-sum, also known as an exclusive scan operation. That is, $d_i = \sum_{j=0}^{i-1} k_j$ and $d_0 = 0$. This can be computed in parallel in $O(\log r)$ elapsed time, although it is already so fast, that it does not matter.

Finally, the array b is allocated, and the triangles are inserted into it. A counter is kept of the number of triangles that have been inserted so far into each cell. That can be stored in cells of b that have not been used so far. This can mostly be done in parallel. The counter for the number of triangles in each cell has to be read and incremented atomically. However, when the number of cells is much more than the number of threads, collisions are rare.

Fig. 2 illustrates the indexing of a mesh using a 2D uniform grid built with a dynamic array 2(a), and a ragged array 2(b).

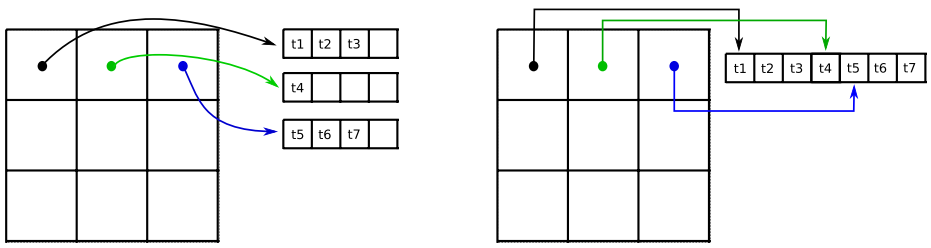


Fig. 2. Dynamic array versus ragged array - 3 × 3 uniform grid using dynamic arrays (a) versus ragged array (b). Only the memory related to the first row of the grid is shown.
Source: [27].

4.2. Bounding-box Culling

For large datasets, depending on r, there may be many triangles in each cell. In each cell, we will doubly iterate through the red and the blue triangles, testing each red–blue pair for intersection. Before the exact test, we use a bounding box cull, since it is much faster than the exact test, and it eliminates many pairs that do not intersect.

Doing this correctly requires care. However that is necessary because an incorrectly determined intersection might cause topological errors when this algorithm is a subroutine in a bigger algorithm such as determining all the intersection information about intersecting two meshes, in addition to just the intersecting triangles. The key is that, whenever a coordinate of a bounding box is approximated, to round it in the direction of making the box bigger.

The bounding-box filtering can be done on the GPU with two passes. In the first pass, the number of pairs whose bounding-boxes intersect each other is counted. Each thread is responsible for checking one pair of triangles. If their bounding-boxes overlap, the thread increments a counter (stored in the GPU global memory) via an atomic add operation (*atomicAdd*). We will show in the experiments that the synchronized operation does not impose a significant performance overhead. If this step were a bottleneck, it could be improved with a more sophisticated reduction operation using a faster memory from the hierarchy.

The counter is returned to the CPU, which then allocates an array to hold all the pairs whose bounding-boxes do overlap, and so have to be tested for intersection. The algorithm then proceeds to the second pass, and the GPU is now responsible for populating this array. Each thread processes again one pair of triangles, and if their bounding-boxes overlap, the thread inserts its pair into the array. The insertion is performed by employing another global counter (initially set to 0), which is also incremented employing the thread-safe *atomicAdd* operation. This operation returns the value of the variable, and then increments it. This ensures each thread will insert a triangle into a different position of the array.

To perform these passes, the pairs that each thread will process have to be carefully selected. Traversing the grid to create an array of pairs so that each GPU thread would be responsible for processing a pair can be unfeasible, as shown in our previous work [3]. In order to avoid this costly pre-processing step, an approach for implicitly associating a thread with a pair will be employed.

4.3. Implicit Thread Association

In [3] we presented three techniques for delegating work to be done by the GPU threads when there are many pairs of objects in a uniform grid. The focus of that work was on intersecting segments against triangles, but this delegation strategy can also be applied for the current problem.

In this paper we will employ the method with the best performance to launch the threads in the two bounding-box filtering passes on the GPU. This subsection will describe this technique (details are available in [3]).

The idea is to launch the thread blocks such that threads in the same block will always process the same uniform grid cell. Let T_{1c} and T_{2c} be the number of triangles in meshes T_1 and T_2, respectively that are contained in the uniform grid cell c. If the algorithm is configured to launch T_c threads per CUDA block, $\lceil \frac{T_{1c} \times T_{2c}}{T_c} \rceil$ blocks will have to be launched to completely process the pairs of triangles in c.

The following three arrays are created in order for each thread to determine which pair of triangles it is responsible for processing. For simplicity, these arrays are initially created on the CPU and then copied to the GPU:

- $Cell[b]$: is the uniform grid cell being processed by block b.
- $First_{pair}[b]$ and $Last_{pair}[b]$: represent the index of, respectively, the first and last pair of triangles being processed by block b.

To determine which pair P (within the cell $Cell[b]$) of triangles the thread with id tid in block b will process, the following expression is employed: $P = tid + First_{pair}[b]$. Given a pair P being processed by a CUDA block b in cell $C = Cell[b]$, then this pair contains the triangles with indices $P\%T_{2c}[C]$ in mesh T_2, and $P/T_2c[C]$ in mesh T_1.

The following examples (adapted from [3]) illustrate how threads are associated with the triangles they are processed:

Table 1 shows the distribution of triangles from two meshes in a uniform grid with 4 cells. For example, cell 0 has 3 triangles from mesh T_{1c} and 4 triangles from cell T_{2c}, and so, 12 pairs of triangles will have to be evaluated for intersection in that cell.

If each CUDA block has 4 threads, then 3 blocks will be required to process cell 0 and 2 blocks will be required to process the 6 pairs of triangles in cell 1. Therefore, 7 blocks will be required to process all the pairs.

Table 2 presents the arrays $Cell$, $First_{pair}$ and $Last_{pair}$ created for the uniform grid from Table 1. For example, a thread in block 1 will process the pairs 4, 5, 6, 7, and thus, the third thread within that block will evaluate pair 6 for intersection (consisting of triangles $6/4 = 1$ from mesh T_1 and $6\%4 = 2$ from mesh T_2).

This strategy balances the amount of work performed by each thread block, since all blocks (except possibly the last) are configured to process the same number of pairs of triangles.

The strategy presented in this subsection is employed to efficiently launch the kernels that count the number of bounding-box overlaps, and also that create the array of pairs to be tested for intersection.

Table 1
Uniform grid cells.
Source: [3].

Cell	0	1	2	3
T_{1c}	3	2	1	3
T_{2c}	4	3	1	1
Blocks	3	2	1	1

Table 2
GPU blocks of threads.
Source: [3].

Block	0	1	2	3	4	5	6
Cell	0	0	0	1	1	2	3
$First_{pair}$	0	4	8	0	4	0	0
$Last_{pair}$	3	7	11	3	5	0	0

The other strategies evaluated in [3] presented challenges such as load unbalancing. For example, if each CUDA thread was configured to process one triangle t from T_1 and test it for intersection against triangles from T_2 in the same uniform grid cells as t, the varying number of triangles in each cell could create thread divergence and load unbalance.

4.4. Intersection evaluation

The main step of the algorithm is evaluating intersection predicates with interval arithmetic. After the uniform grid indexing and bounding-box culling, the result is an array (already located on the GPU) of potentially intersecting triangle pairs. These pairs are then tested for intersection.

Two additional arrays are allocated in the GPU (with the same size as the array of pairs) in order to store the pairs of triangles that intersect, and also the ones that resulted in interval failures.

After the arrays are allocated, a GPU kernel is launched to test the pairs for intersection. Each thread evaluates the intersection of a pair of triangles (t_1, t_2), using five 3D orientation predicates in order to verify if any edge of t_1 intersects t_2 (and vice-versa) [31].

The intersecting pairs (and the ones whose intersections resulted in interval failures) are inserted into the resulting arrays by employing a global counter for each array, using thread-safe *atomicAdd* operations.

A triangle may overlap, and so be inserted into, two or more different uniform grid cells. Therefore one pair of triangles may be tested for intersection more than once, causing duplicates in the intersections array. These duplicates are removed in the next step.

4.5. Eliminating Duplicates and Performing Exact Re-evaluation

The resulting arrays of intersecting pairs of triangles and interval failures are sorted on the GPU with a parallel radix-sort, and then the duplicates are removed. This uses *sort* and *unique* from the *Thrust* library [32]. (When the sort key is a plain old datatype (POD), the radix sort takes time linear in the number of records being sorted. It also reads and writes in a predictable pattern. Both properties contrast favorably to the widely used quicksort.) The sorted arrays are then copied back to the CPU for possible future use by other algorithms.

In the next step, the array containing the pair of triangles whose intersection computation resulted in interval failures is exactly re-evaluated on the CPU in parallel using OpenMP. Each pair is evaluated using the same predicates that were used on the GPU. However this time, the computation is performed with arbitrary precision rational numbers instead of floating-point intervals.

If the intervals are implemented using single-precision floating-point numbers, the post-processing time can be reduced by re-evaluating the arithmetic failures with double-precision, and then using the rationals only for the predicates that also led to failures with doubles. This idea of re-evaluating predicates with increasing precision in geometric predicates has been previously applied, for example, by Shewchuck [11].

The intersections detected in this step are then appended to the ones detected on the GPU.

4.6. Algorithm complexity

The complexity of the bounding-box culling step is $O(r^3 + B_t)$, where r^3 is the number of cells (for a uniform grid resolution r) and B_t is the total number of bounding-box tests in all grid cells.

From Section 4.1, B_t is expected to be linear in the size of the input plus the number of intersecting bounding-boxes. Thus, $B_t = O(T_1 + T_2 + I)$, where T_1 and T_2 are the number of triangles in the input meshes and I is the number of intersecting bounding boxes in all grid cells.

In the worst case, the intersection tests and post-processing steps will have a time complexity $O(I)$ since the parallel sort in the post-processing step is linear complexity, and the number of operations performed on each pair of triangles is constant.

Therefore, the total time and space complexity of the algorithm are both $O(r^3 + T_1 + T_2 + I)$. The resolution of the grid (r) is a small constant tuned by the user to achieve a small number of triangles per cell while keeping the number of duplicate bounding boxes in the grid small. That way, I is linear on the number of unique intersecting bounding boxes.

An adversary could create hard test cases where all the triangles in the two input meshes are equal, or every bounding-box intersects every cell. Then each triangle will be in all r^3 cells, and thus, $I = O(r^3 \times T_1 \times T_2)$, while the number of unique intersections is $O(T_1 \times T_2)$. However, bad test cases could also be created for other algorithms, such as the plane-sweep. Finally, this does not affect our parallel speedup, since the number of intersection tests would be equal to the one in the sequential algorithm.

5. Experiments

The fast algorithm for intersecting triangles was implemented in C++ and CUDA. It was evaluated on a dual Intel Xeon E5-2660 CPU at 2 GHz (3.2 GHz Turbo Boost), with 256 GB of RAM and a NVIDIA Quadro RTX 8000 GPU. Arbitrary-precision arithmetic was provided by the GMP library [17].

We always used a uniform grid of size $100 \times 100 \times 100$. However, there are heuristics for automatically choosing a grid resolution basing on statistics of the input datasets [27,33], such as choosing the grid size make the expected number of edges per cell a constant. As shown by Magalhães et al. [27], the range of grid configurations with reasonable performance optimum is broad. This follows because a finer grid causes one part of the algorithm to run more quickly and another part to run more slowly.

We performed experiments on meshes from two public repositories. The *Armadillo* mesh was downloaded from the Stanford repository [34] and the other ones were downloaded from Thingi10K [35]. These meshes were first tetrahedralized using GMSH [36]. Table 3 describes the dataset and Fig. 3 illustrates it. These datasets' sizes ranged from $600K$ to $8M$ triangles. Fig. 4 shows the interior of mesh *OpenToys* after tetrahedralization, and Fig. 5 shows two overlaid pairs of meshes employed in the experiments.

In contrast to CPUs, GPUs typically have significantly more single-precision floating point units than double-precision (since they focus on applications where the lower precision of floats is acceptable) [37]. For example, the RTX 8000 GPU employed

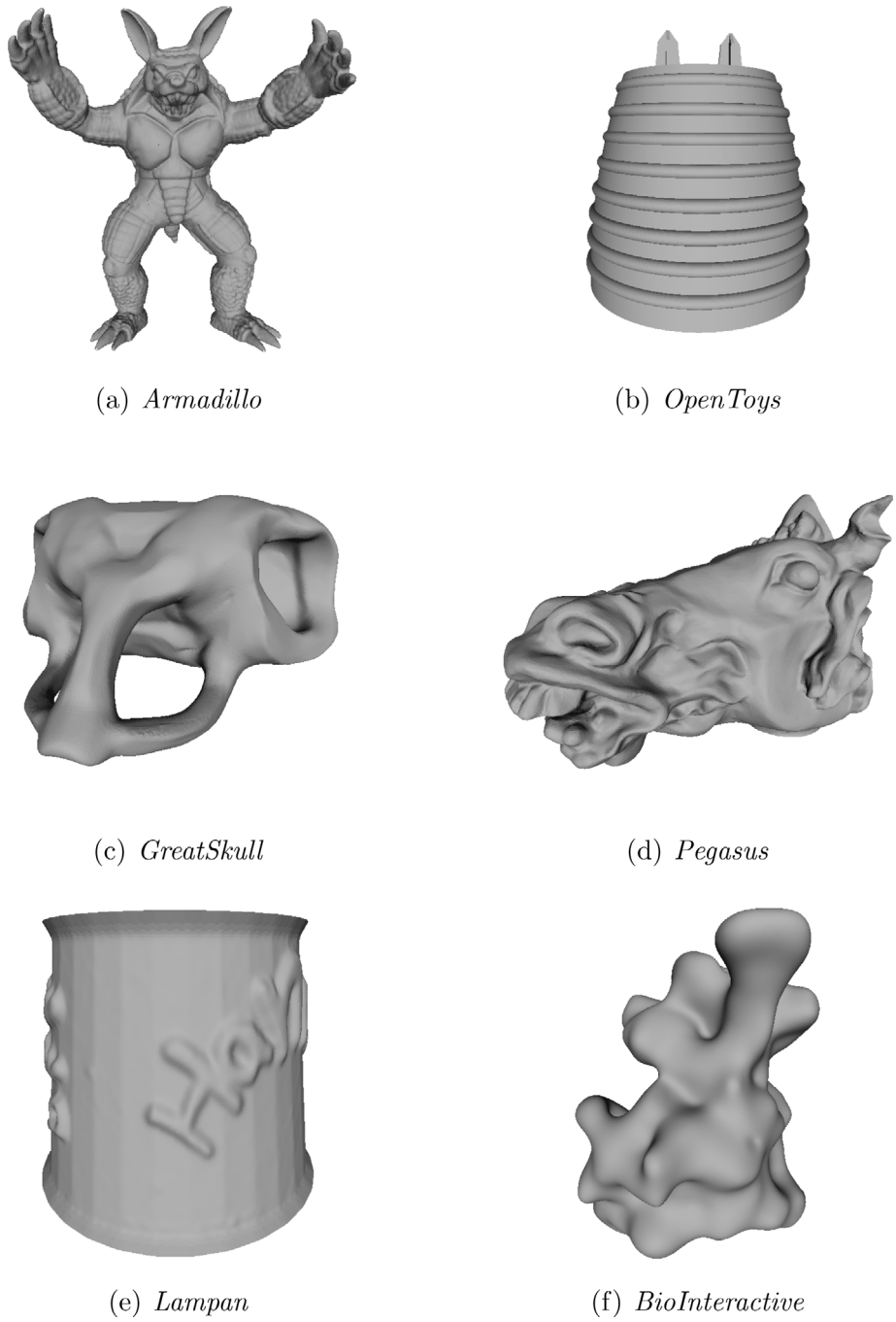


Fig. 3. Meshes employed in the experiments.

Table 3
Number of vertices and triangles in the meshes employed in the experiments. The mesh id can be used to locate the ones downloaded from the Thingi10K website.

Mesh name	Mesh id	# vertices	# triangles
OpenToys	914686	66 166	605 279
GreatSkull	260537	67 265	664 101
Pegasus	68380	106 955	1 066 547
Armadillo	–	340 043	3 377 086
Lampan	518092	603 116	5 937 604
BioInteractive	461112	841 883	8 494 878

in these experiments has a peak single and double-precision performance of, respectively, 16 TFLOPS and 0.5 TFLOPS. Thus we also implemented a version of the GPU code employing intervals with single-precision floats. As described in Section 4.5, the single-precision arithmetic failures were re-evaluated with double-precision intervals, and finally only if necessary were re-evaluated with rationals.

The following listing contains the 3 implementations evaluated in the experiments:

- *CPU*: sequential CPU implementation employing arithmetic filters using double-precision intervals (provided by the CGAL Interval_nt class [16]). This is the baseline implementation.
- *GPUDouble*: parallel GPU implementation using double-precision intervals implemented as described in this paper.

Table 4
Times (in seconds) spent by the intersection tests between meshes *GreatSkull* vs *OpenToys* and *Pegasus* vs *OpenToys*. *Speedup* shows the speedup of the corresponding GPU version of the algorithm when compared against the CPU one.

Dataset	*GreatSkull* vs *OpenToys*				
Method:	CPU	GPUDouble		GPUFloat	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
Pre-processing	1.13	0.05	25	0.05	25
Intersection	55.59	1.09	51	0.04	1496
Post-processing	0.89	0.02	51	0.03	30
Data transfer	-	0.14	—	0.14	—
Total time	57.62	1.29	45	0.25	232
#bounding-box tests	109.1×10^6				
#unique bounding-box tests	33.5×10^6				
#intersection tests	59.5×10^6				
#intersections	2.2×10^6				
Dataset	*Pegasus* vs *OpenToys*				
Method:	CPU	GPUDouble		GPUFloat	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
Pre-processing	1.73	0.07	24	0.07	25
Intersection	81.59	1.63	50	0.05	1579
Post-processing	1.40	0.04	31	0.04	33
Data transfer	-	0.25	—	0.24	—
Total time (s)	84.71	2.00	42	0.41	208
#bounding-box tests	152.1×10^6				
#unique bounding-box tests	44.2×10^6				
#intersection tests	88.6×10^6				
#intersections	3.2×10^6				

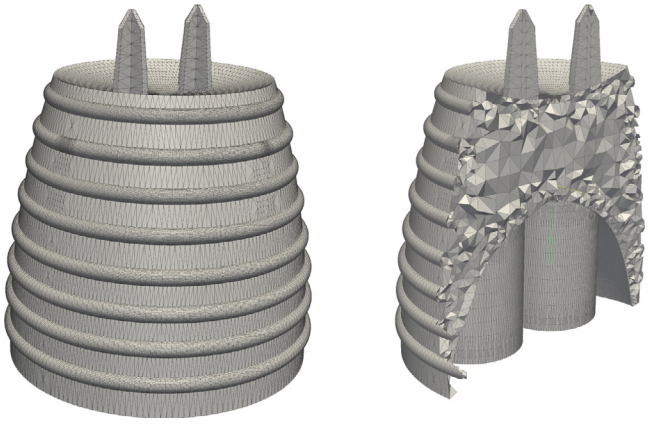


Fig. 4. Mesh *OpenToys* (left). The right figure illustrates its tetrahedralized interior.
Source: [27].

- *GPUFloat*: the same implementation as GPU double, but with single-precision floats in the intervals.

5.1. Experiments on the RTX 8000 GPU

Tables 4 and 5 present the results of intersecting pairs of triangles from the input meshes. We assume the uniform grid index is already created as a ragged-array and loaded in the CPU memory.

Separate times were measured for pre-processing, intersection, post-processing, and data transfer.

The pre-processing includes accessing the index and performing bounding-box tests to cull the number of triangles actually tested for intersection. For the GPU implementation, the time for processing the uniform grid to distribute the work to the threads (as described in Section 4.3) is also included (see Section 4.3).

The intersection is the time spent evaluating pairs of triangles for intersection using interval arithmetic. In the current implementation, our algorithm only reports nondegenerate intersections.

Special cases (such as two triangles intersecting at a single point), which happen when at least one of the orientation predicates returns 0, could be handled by flagging these cases and re-evaluating them on the CPU during the post-processing. Indeed, interval arithmetic would typically lead to interval failures in these cases because the sign of the interval of a determinant that is equal to 0 cannot be reliably determined (since floating-point errors will usually cause its bounds not to be both 0). This is not a limitation of the proposed technique for two reasons: first, this affects other algorithms based on interval arithmetic, and second, as will be shown in the experiments, the number of intersection tests is typically much larger than the number of intersections.

Post-processing includes sorting the pairs of intersecting triangles to remove duplicates, and then re-evaluating, on the CPU, the computations whose intervals failed.

The data transfer is the total time for transferring the input, intermediate and output data between the CPU and GPU.

Finally, the last four rows of each table include the number of bounding-box tests performed by the algorithm (including duplicates), the number of unique bounding-box tests , the number of intersection tests (i.e., the number of pairs of triangles whose bounding-boxes do intersect) and the number of intersections.

In the *CPU* version of the algorithm, the bottleneck in all cases was testing pairs of triangles for intersection using interval arithmetic. In this step, *GPUDouble* achieved a speedup ranging from 50× to 63×, while *GPUFloat* achieved up to 1936× of speedup. This performance difference is consistent with the difference in the peak single and double precision floating-point computing powers of the RTX 8000 GPU.

The good performance of the GPU algorithms in this step can be explained because they are compute-intensive, using 3*D* vector operations such as subtraction and mixed-product.

In all tests with *GPUFloat*, the bottleneck was the data transfer. More than 80% of the total data transfer time was used for initially sending the uniform grid, the triangles and the auxiliary

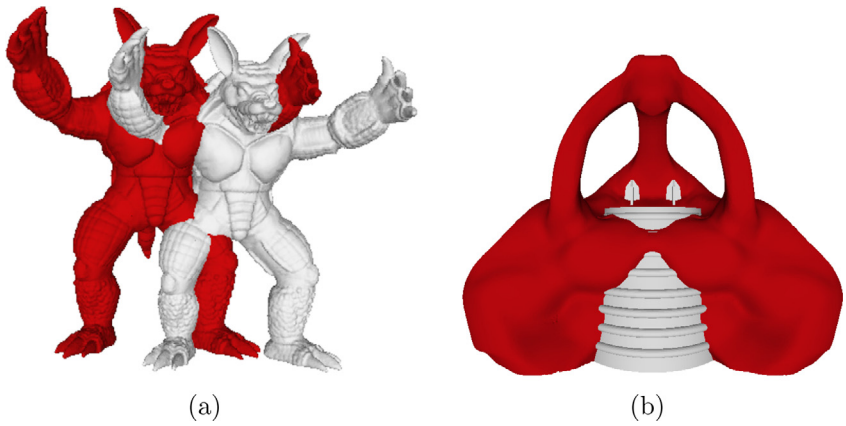


Fig. 5. Examples of pairs selected for intersection tests. (a) *Armadillo* (red) and *ArmadilloTranslated* (white) (b) *GreatSkull* (red) and *OpenToys* (white).

Table 5
Times (in seconds) spent by the intersection tests between meshes *Armadillo* vs *ArmadilloTranslated* and *BioInteractive* vs *Lampan*. *Speedup* shows the speedup of the corresponding GPU version of the algorithm when compared against the CPU one.

Dataset	*Armadillo* vs *ArmadilloTranslated*				
Method:	CPU	GPUDouble		GPUFloat	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
Pre-processing	2.48	0.14	18	0.14	18
Intersection	67.49	1.33	51	0.03	1936
Post-processing	1.05	0.06	17	0.06	18
Data transfer	-	0.32	–	0.33	–
Total time (s)	71.03	1.86	38	0.56	127
#bounding-box tests	339.0×10^6				
#unique bounding-box tests	215.0×10^6				
#intersection tests	72.8×10^6				
#intersections	6.8×10^6				
Dataset	*BioInteractive* vs *Lampan*				
Method:	CPU	GPUDouble		GPUFloat	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
Pre-processing	7.11	0.28	25	0.28	25
Intersection	474.56	7.56	63	0.32	1462
Post-processing	1.33	0.02	72	0.05	25
Data transfer	-	0.56	–	0.51	–
Total time (s)	482.99	8.42	57	1.17	414
#bounding-box tests	655.9×10^6				
#unique bounding-box tests	273.9×10^6				
#intersection tests	505.4×10^6				
#intersections	5.3×10^6				

arrays used for the implicit thread association to the GPU. Indeed, excluding the data transfers, the best total speedup would have been 731× instead of 414×. Thus, applications not requiring frequent data transfer between the CPU and GPU could benefit even further from our method. Also, if processing many datasets simultaneously, one job's data transfer could occur concurrently with another job's computation, perhaps using CUDA streams.

The worst total speedup of *GPUFloat* was 127×, with the *Armadillo* datasets. This step had the lowest percentage of pairs of triangles (selected by the index) whose bounding-boxes do intersect. Indeed, only 21% of the pairs of bounding-boxes filtered by the uniform grid do intersect, as opposed to 55%, 58% and 77% (the best total speedup) in the other test cases. As a result, in this dataset 21% of the pairs of triangles selected at the indexing step reached the intersection computation step (the most time-consuming one in the CPU and the one with the best parallel speedup), which reduces the total speedup.

The two GPU implementations had similar performance in the data transfer and pre-processing steps (in the worst case *GPUDouble* was 10% slower than *GPUFloat*). The higher post-processing time of *GPUFloat* can be explained because of the lower precision of floats, which leads to more interval failures, requiring more intersections to be re-evaluated with doubles and rationals on the CPU.

In the experiments intersecting meshes *BioInteractive* and *Lampan*, for example, the post-processing time of 0.05 s for *GPUFloat* was composed of 0.011*s* for sorting the resulting intersections and removing duplicates, 0.003 s for filtering the pairs of triangles whose intersection computations had interval failures and 0.039 s for re-evaluating the interval failures with doubles. In this experiment, only 0.01% of the intersection tests led to failures and the results with double precision were computed successfully (thus, rationals were not necessary). *GPUDouble* and *CPU*, on the other hand, had no filter failures and, thus, only spent time sorting the results to remove duplicates.

The number of pairs of bounding-boxes evaluated for intersection was from 1.6 to 3.4 times larger than the number of unique pairs. These duplicates are not removed at the beginning of the algorithm because, as mentioned in Section 4.3, the list of pairs is implicitly created by the threads. Since the performance bottleneck is data transfer and these duplicates are never transferred

Table 6
Times (in seconds) spent by the different versions of the algorithms for intersecting a particular pair of meshes. The methods labeled with * do not employ a bounding-box culling. GPUD. and GPUF. represent, respectively, the *GPUDouble* and *GPUFloat* methods.

Dataset	*GreatSkull* vs *OpenToys*					
Method	CPU	CPU*	GPUD.	GPUD.*	GPUF.	GPUF.*
Pre-processing	1.13	1.14	0.05	0.04	0.05	0.04
Intersection	55.59	100.50	1.09	2.03	0.04	0.05
Post-processing	0.89	0.90	0.02	0.02	0.03	0.04
data transfer	—	—	0.14	0.14	0.14	0.14
Total time	57.62	102.54	1.29	2.22	0.25	0.26

Table 7
Times (in seconds) spent by the different versions of the algorithms for intersecting a particular pair of meshes. The GPU algorithms were evaluated on a GeForce GTX 1070 Ti GPU. *Speedup* shows the speedup of the corresponding GPU version of the algorithm when compared against the CPU version.

Dataset	*Pegasus* vs *OpenToys*				
Method:	CPU	GPUDouble		GPUFloat	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
Pre-processing	1.24	0.09	14	0.10	13
Intersection	82.24	2.62	31	0.20	407
Post-processing	1.39	0.04	32	0.04	31
Data transfer	—	0.19	—	0.19	—
Total	84.86	2.94	29	0.54	158

to the CPU, this did not significantly affect the efficiency of the algorithm.

Over all the experiments, in the worst case (intersection of meshes *Pegasus* and *OpenToys*), only 0.001% of the intersection tests had to be performed with rationals.

Table 6 compares the 3 implementations against versions without the bounding-box culling. As can be seen, detecting intersections was 80%, 86% and 25% slower when no bounding-box filtering was performed in, respectively, the *CPU*, *GPUDouble* and *GPUFloat* methods, while there was no significative difference in the other steps. As shown in Tables 4 and 5, in this test case, the number of intersection tests when no bounding-box filtering is employed was 83% larger (109×10^6 versus 59.5×10^6).

5.2. Experiments on a lower-end computer

We also performed experiments on another computer with an NVIDIA 1070 Ti GPU (8 GB of RAM), AMD Ryzen 5 1600 CPU 3.2 GHz (12 hyperthreads), 16 GB of RAM in order to evaluate the proposed technique on a lower-end GPU. Table 7 presents the results for a pair of meshes. The biggest difference was in the intersection step, whose time increased from 0.05 s to 0.20s compared to the time on the RTX 8000 GPU. Because of the data transfer operations and sequential steps of the algorithm, the difference in the total running-time (for *GPUFloat*) is smaller (the total time increased from 0.41 s to 0.54 s on the lower-end GPU).

Even though, as expected, the running-time on the lower-end GPU is higher than on the RTX 8000 GPU, the performance difference when compared against the CPU implementation is still high. For example, *GPUFloat* achieved speedups of, respectively, 407× and 158× for the intersection and total times when compared against *CPU*.

6. Conclusions and future work

We proposed the use of GPUs to accelerate the evaluation of exact geometric predicates filtered with intervals of floating-point numbers. The idea is to evaluate the predicates using interval arithmetic on the GPU. The few results that could not be guaranteed to be correct are then re-evaluated on the CPU using arbitrary-precision rationals.

As a proof of concept, a parallel algorithm for detecting intersections of red and blue triangles has been implemented. Because of the high computing power of the GPU for processing floating-point numbers, a speedup of up to 1936 times (when compared against the sequential version) was obtained in the evaluation of the predicates. The speedup of the algorithm was up to 414 times if the total running-time was considered.

Even though the time analysis assumed i.i.d input, the algorithm's good performance on the decidedly nonuniform test cases demonstrates its general utility.

This performance and exactness make this technique applicable for interactive applications, e.g., in CAD, GIS, computational geometry, and 3D modeling.

In the future, we intend to apply this technique to other problems such as convex hull computation, 2D and 3D point location and boolean operations on meshes. Applications whose bottleneck is the evaluation of predicates could particularly present a better speedup.

Also, we intend to further improve the performance of the predicates. For example, the communication between the CPU and the GPU causes a significant overhead. Moving the combinatorial part of the algorithms to the GPU might help. Unified memory, where the GPU automatically migrates the data where needed, might be useful.

Finally, testing this technique on other architectures would also be useful: for example, high-end Xeon processors are MIMD (*Multiple Instruction, Multiple Data*) processors (making it easier to port the combinatorial components of the algorithms to them). At the same time, these devices have a high parallel computing power for processing floating-point numbers (thanks to wide *Single Instruction, Multiple Data* - SIMD instructions in the individual cores). Thus, we believe both algorithms and exact geometric predicates could be accelerated on these devices using these instructions (keeping both in the same device would reduce the communication overhead).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was partially supported by CAPES, FAPEMIG and FUNARBE, Brazil.

References

[1] Brönnimann Hervé, Burnikel Christoph, Pion Sylvain. Interval arithmetic yields efficient dynamic filters for computational geometry. Discrete Appl Math 2001;109(1–2):25–47.
[2] de Matos Menezes Marcelo, Magalhães Salles Viana Gomes, Franklin W Randolph, de Oliveira Matheus Aguilar, Chichorro Rodrigo EO Bauer. Accelerating the exact evaluation of geometric predicates with GPUs. In: Shontz Suzanne, Peiró Joaquim, Viertel Ryan, editors. 28th international meshing roundtable. Buffalo, NY, USA; 2019, http://dx.doi.org/10.5281/zenodo.3653101.
[3] Menezes Marcelo, V. G. Magalhães Salles, Aguilar Matheus, Franklin W Randolph, Coelho Bruno. Employing GPUs to accelerate exact geometric predicates for 3D geospatial processing. In: Krumm John, editor. 2nd ACM SIGSPATIAL international workshop on spatial gems. ACM; 2020, URL https://www.spatialgems.net/.
[4] European Space Agency. Ariane 501 inquiry board report. 2015, Retrieved on 06/15/2015 URL.
[5] Skeel Robert. Roundoff error and the Patriot missile. SIAM News 1992;25(4):11.

[6] Kettner Lutz, Mehlhorn Kurt, Pion Sylvain, Schirra Stefan, Yap Chee Keng. Classroom examples of robustness problems in geometric computations. Comput Geom 2008;40(1):61–78. http://dx.doi.org/10.1016/j.comgeo.2007.06.003.
[7] Hobby John D. Practical segment intersection with finite precision output. Comput Geom 1999;13(4):199–214.
[8] de Berg Mark, Halperin Dan, Overmars Mark. An intersection-sensitive algorithm for snap rounding. Comput Geom 2007;36(3):159–65.
[9] Hershberger John. Stable snap rounding. Comput Geom 2013;46(4):403–16.
[10] Belussi Alberto, Migliorini Sara, Negri Mauro, Pelagatti Giuseppe. Snap rounding with restore: An algorithm for producing robust geometric datasets. ACM Trans Spatial Algorithms Syst 2016;2(1):1:1–36. http://dx.doi.org/10.1145/2811256.
[11] Shewchuk Jonathan Richard. Adaptive precision floating point arithmetic and fast robust geometric predicates. Discret Comput Geom 1997;18(3):305–63.
[12] Li Chen, Pion Sylvain, Yap Chee Keng. Recent progress in exact geometric computation. J Log Algebr Program 2005;64(1):85–111. http://dx.doi.org/10.1016/j.jlap.2004.07.006.
[13] Hoffman Christoff M. The problems of accuracy and robustness in geometric computation. Comput 1989;22(3):31–40.
[14] Yap Chee Keng. Towards exact geometric computation. Comput Geom 1997;7(1–2):3–23.
[15] Pion Sylvain, Fabri Andreas. A generic lazy evaluation scheme for exact geometric computations. Sci Comput Program 2011;76(4):307–23.
[16] The CGAL Project. Cgal, computational geometry algorithms library. 2015, http://www.cgal.org (Retrieved on 10/19/2017).
[17] Granlund Torbjorn, the GMP development team. GNU MP: The GNU multiple precision arithmetic library, 6th ed.. 2014, http://gmplib.org/ (Retrieved on 10/19/2017).
[18] Jacobson Alec, Panozzo Daniele, et al. Libigl: A simple c++ geometry processing library. 2016, http://libigl.github.io/libigl/ (Retrieved on 10/18/2017).
[19] Audet Samuel, Albertsson Cecilia, Murase Masana, Asahara Akihiro. Robust and efficient polygon overlay on parallel stream processors. In: Proc. 21st ACM SIGSPATIAL int. conf. advances geographic information systems. SIGSPATIAL'13, New York, NY, USA: ACM; 2013, p. 304–13. http://dx.doi.org/10.1145/2525314.2525352.
[20] Magalhães Salles VG, Franklin W Randolph, Andrade Marcus VA. Fast exact parallel 3D mesh intersection algorithm using only orientation predicates. In: Proceedings of the 25th ACM SIGSPATIAL international conference on advances in geographic information systems. ACM; 2017, p. 44.
[21] Popescu Valentina. Towardss fast and certified multiple-precision librairies. (Ph.D. thesis), Université de Lyon; 2017.
[22] Joldes Mioara, Muller Jean-Michel, Popescu Valentina, Tucker Warwick. CAMPARY: CUDA multiple precision arithmetic library and applications. In: International congress on mathematical software. Springer; 2016, p. 232–40.
[23] Lu Mian, He Bingsheng, Luo Qiong. Supporting extended precision on graphics processors. In: Proceedings of the sixth international workshop on data management on new hardware. ACM; 2010, p. 19–26.
[24] Collange Sylvain, Daumas Marc, Defour David. Chapter 9 - interval arithmetic in CUDA. In: mei W. Hwu Wen, editor. GPU computing gems jade edition. Applications of GPU computing series, Boston: Morgan Kaufmann; 2012, p. 99–107. http://dx.doi.org/10.1016/B978-0-12-385963-1.00009-5.
[25] Collange Sylvain, Flórez Jorge, Defour David. A GPU interval library based on boost.interval. In: 8th conference on real numbers and computers. 2008, p. 61–71.
[26] Whitehead Nathan, Fit-Florea Alex. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Rn (A+ B) 2011;21(1):18749–9424.
[27] Magalhães Salles VG, Franklin W Randolph. Exact and parallel intersection of 3d triangular meshes. (Ph.D. thesis), Rensselaer Polytechnic Institute, USA; 2017.
[28] Akman V, Franklin Wm Randolph, Kankanhalli Mohan, Narayanaswami Chandrasekhar. Geometric computing and the uniform grid data technique. Comput Aided Des 1989;21(7):410–20.
[29] Franklin Wm Randolph, Chandrasekhar Narayanaswami, Kankanhalli Mohan, Seshan Manoj, Akman Varol. Efficiency of uniform grids for intersection detection on serial and parallel machines. In: Magnenat-Thalmann Nadia, Thalmann D, editors. New trends in computer graphics. Berlin, Germany: Springer-Verlag; 1988, p. 288–97.
[30] Hopkins Sara, Healey Richard G. A parallel implementation of Franklin's uniform grid technique for line intersection detection on a large transputer array. In: Brassel Kurt, Kishimoto H, editors. 4th int. symp. spatial data handling. Zürich; 1990, p. 95–104.
[31] Segura Rafael Jesús, Feito Francisco R. Algorithms to test ray-triangle intersection. Comparative study. In: The 9-Th int. conf. central europe comput. graph., visualization comput. vision'2001. 2001, p. 76–81.
[32] Hoberock Jared, Bell Nathan. Thrust: A parallel template library. 2010, http://thrust.github.io/ (Retrieved on 10/19/2017).
[33] Audet Samuel, Albertsson Cecilia, Murase Masana, Asahara Akihiro. Robust and efficient polygon overlay on parallel stream processors. In: Proceedings of the 21st ACM SIGSPATIAL international conference on advances in geographic information systems. ACM; 2013, p. 304–13.
[34] The Stanford 3D Scanning Repository. "The stanford 3D scanning repository". 2016, URL http://graphics.stanford.edu/data/3Dscanrep/ (Retrieved on 10/19/2017).
[35] Zhou Qingnan, Jacobson Alec. Thingi10k: A dataset of 10,000 3D-printing models. 2016, arXiv preprint arXiv:1605.04797.
[36] Geuzaine Christophe, Remacle Jean-François. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. Int J Num Methods Eng 2009;79(11):1309–31. http://dx.doi.org/10.1002/nme.2579.
[37] Sun Yifan, Agostini Nicolas Bohm, Dong Shi, Kaeli David. Summarizing CPU and GPU design trends with product data. 2019, arXiv preprint arXiv:1911.11313.