Fast parallel evaluation of exact geometric predicates on GPUs

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

[a] *Universidade Federal de Viçosa (MG) Brasil, marcelo.menezes@ufv.br*
[b] *Universidade Federal de Viçosa (MG) Brasil, salles@ufv.br*
[c] *Universidade Federal de Viçosa (MG) Brasil, matheus.a.aguilar@ufv.br*
[d] *Rensselaer Polytechnic Institute, Troy NY, USA 12180, mail@wrfranklin.org*
[e] *Universidade Federal de Viçosa (MG) Brasil, rodrigo.chichorro@ufv.br*

Abstract

This paper presents a technique for employing high-performance computing for accelerating the exact evaluation of geometric predicates. Arithmetic filters are implemented using interval arithmetic to reduce the necessity of exact arithmetic while ensuring the results of the predicates are still exact. Furthermore, the computation with interval arithmetic is offloaded to a CUDA-enabled GPU. If the GPU detects that some results cannot be trusted, the corresponding predicates are re-evaluated in parallel on the CPU using arbitrary-precision rational numbers. As a case study, a red-blue 3D triangle intersection algorithm has been implemented. Since the intervals are implemented using floating-point numbers, the parallel computing power of GPUs for processing these numbers led to a speedup of up to 1936 times (when compared against a similar sequential implementation) in the evaluation of these predicates (and up to 414 times if the entire runnning-time of the algorithm is considered). The excellent performance associated to the exactness makes this technique suitable for accelerating geometric operations in fields such as CAD, GIS and 3D modeling.

Keywords: Boolean operations; Parallel programming; Exact computation; Polyhedron intersection

1. Introduction

A particular challenge in computational geometry problems is to address the errors caused by floating-point arithmetic. Inexact floating-point numbers violate most of the axioms of an algebraic field. For example, addition is not associative. Roundoff errors cause 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 propagate to higher-level operations (such as using orientation predicates to compute a convex hull), what makes the design of correct algorithms even harder.

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

A technique to guarantee computation will be free from round-off errors is representing the coordinates with exact arbitrary-precision rational numbers. The drawback is that in some applications the overhead associated to these numbers may be unacceptable. Also, the number of digits in the numerator and denominator of these numbers grow as arithmetic operations are performed (the size is typically the sum of the number of digits in the operands) and, thus, performance may degrade if the computation tree is deep.

Some techniques have been proposed to cope with this performance problem. Namely, arithmetic filters using interval arithmetic represent each exact number e as an interval of floating-point values containing e. Thanks to guarantees of the IEEE-754 floating-point standard, for each arithmetic operation a new interval (which is guaranteed to contain the exact result of that operation) can be computed. Thus, predicates can be initially evaluated using intervals. If it is detected that the exact result of that predicate can be inferred from the bounds of the interval, this result is computed. Otherwise, the expression is re-evaluated using exact arithmetic (or intervals with more precise number types). As mentioned in [5], most of the time computation with intervals is enough to infer the exact result and, thus, predicates can be efficiently and exactly evaluated without the overhead of exact computation.

While recently the computing capabilities of desktop computers and workstations have increased due to multi-core processors and accelerators such as GPGPUs (*General Purpose Graphics Processing Unit*) and MICs (*Many Integrated Core Architecture*), many algorithms are still designed considering sequential architectures and, thus, they cannot take advantage of this computing power.

In this paper, we propose the use of a combination of GPUs and multi-core CPUs to accelerate the evaluation 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 filtered and re-evaluated in parallel on the CPU using exact arithmetic.

A preliminary version of this framework was implemented and tested for computing the intersections between two sets of 2D segments [9]. The algorithm 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. A list of results is then returned to the CPU. These results are represented using a flag with 3 possible values: intersection, no intersection or uncertain result (this is known as interval failure and it means this intersection cannot be safely determined using intervals). The CPU traverses this list and, for each interval failure, it computes the intersection using multiple-precision rational numbers. When compared against a sequential implementation, the algorithm achieved speedups of up to 289 times for intersection evaluation (and up to 40 times considering the entire running time) on a NVIDIA GTX 1070 Ti GPU.

In [26], we extended the previous algorithm for computing intersections between segments and triangles in 3D. However, a trivial extension did not yield a good performance, once the pre-processing step, where the CPU creates a list of pairs (of segments and triangles), became the bottleneck. Thus, in [26] we proposed a new method for associating the GPU threads with pairs to be tested. 3 auxiliary arrays are employed in order for each thread to determine the pair it is responsible for processing.

This implementation led to a speedup of up to 17 times when compared against a sequential implementation (on a NVIDIA GTX 1070 Ti GPU). 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 a bounding-box culling step that has been added to this algorithm before the intersection tests: this improved its performance, but the improvement in the sequential version was better than in the parallel one (since the bounding-box test creates more thread divergence on the GPU).

Finally, in this paper, we extended the framework even further, improving the bounding-box step to increase the parallel speedup and, also, creating

the array of pairs on the GPU. Furthermore, another improvement presented in this paper is the usage of single-precision floating-point arithmetic (which typically present higher performance than double-precision on GPUs) for the intervals (the unreliable results are re-evaluated with double-precision and, if they are still unreliable, with rationals).

These novel ideas are evaluated in a new case study: detecting the pairwise intersections of triangles (in 3D) from two meshes, which is described in section 4. Experiments have been performed on a faster NVIDIA RTX 8000 GPU and we observed a speedup of up to 1936 times for intersection evaluation (compared against a sequential implementation), and up to 414 times for the entire running time (the intersection and total speedups on the lower-end GTX 1070 Ti GPU were, respectively, 407× and 158×).

The performance and correctness make this technique suitable, for example, for processing large datasets (where the chance of failure in inexact algorithms is higher) in interactive applications such as GIS and CAD systems.

2. Background

2.1. Roundoff errors

Non-integer numbers are typically approximately represented in computers with floating-point values. The difference between the value of a non-integer number and its approximation is often referred as roundoff error. Even though these differences are usually small, these errors accumulate as sequences of arithmetic operations are performed. The presence of floating point errors in computer programs often creates serious consequences in diverse fields such as the failure of the first Ariane V rocket [10] and the failure of the Patriot missile defense system [32].

In geometry, roundoff errors can generate topological inconsistencies causing globally impossible results. For example, if the point of intersection of two lines segments is computed, the result may not lie in any of the two lines. Kettner et al. [21] presented some examples of failures caused by roundoff errors in computational geometry problems. In this study, they presented examples of how the evaluation of orientation predicates can be affected by floating-point errors. As a result, algorithms (such as one for computing convex hulls) relying on these predicates may fail.

The planar orientation predicate is the problem of finding whether three points $p = (p_x, p_y)$, $q = (q_x, q_y)$, $r = (r_x, r_y)$ are collinear, make a left turn,

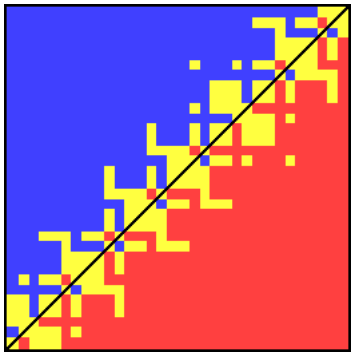


Figure 1: Roundoff errors in the planar orientation problem - Geometry of the planar orientation predicate for double precision floating point arithmetic. Yellow, red and blue points represent, respectively, collinear, negative and positive orientations. The diagonal line is an approximation of the segment (q, r). Source: [21].

or make a right turn. This predicate is computed by evaluating 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 make the sign of this determinant to be evaluated wrongly, mis-classifying the orientation. To illustrate this problem, Kettner et al. [21] implemented a program to apply the planar orientation predicate ($orientation(p, q, r)$) on a point $p = (p_x + xu, p_y + yu)$ where u is the step between adjacent floating point numbers in the range of p and $0 \leq x, y \leq 255$. This results in a 256×256 matrix containing either blue, yellow and red points meaning that the corresponding point is detected to be above, on or below the line that passes through q and r. Figure 1 shows the geometry of this experiment for $p = (0.5, 0.5)$, $u = 2^{-53}$, $q = (12, 12)$ and $r = (24, 24)$. As it can be seen, several points have their orientation computed incorrectly.

As shown by [21], these inconsistent results in the orientation predicates could make algorithms that use this predicate to fail.

Some techniques have been proposed to handle this problem. The simplest one, the epsilon-tweaking, consists of using an ϵ tolerance that considers two values x and y are equal if $|x - y| \leq \epsilon$. However this is a formal mess

because equality is no longer transitive, nor invariant under scaling. Thus, in practice, epsilon-tweaking fails in several situations [21].

Snap rounding is another method to approximate arbitrary precision segments into fixed-precision numbers [15]. However, Snap rounding can generate inconsistencies and deform the original topology if applied consecutively on a data set. Some variations of this technique attempt to get around these issues [8, 14, 4].

Shewchuk [30] presents the Adaptive Precision Floating-Point technique for exactly evaluating predicates. The idea is to perform this evaluation using the minimum amount of precision necessary to achieve correctness. As a result, it is possible to develop some efficient exact geometric algorithms. Geometric predicates can often be evaluated by computing the sign of a determinant and, thus, the actual value of this determinant does not need to be exactly computed as long as the sign of the approximated result is guaranteed to be correct. To determine if the sign of an approximation can be trusted, the approximation and an error estimate are computed and, if the error is big enough to make the sign possibly incorrect, the values are recomputed using higher precision. As mentioned by Shewchuk [30], this technique is not suitable to solve all geometric problems. For example, "a program that computes line intersections requires rational arithmetic; an exact numerator and exact denominator must be stored" [30].

The formally proper way to effectively eliminate roundoff errors and guarantee algorithm robustness is to use exact computation based on rational number with arbitrary precision [22, 17, 21, 38]. Computing in the algebraic field of the rational numbers over the integers, with the integers allowed to grow as long 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 in the result of 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, but that is rarely possible. However, this behavior is acceptable if the depth of the computation tree is small. Also, the performance penalty associated with rationals 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 [27]. The idea is to use an

interval of floating-point numbers containing each exact value. During the evaluation of predicates (which typically consists in the computation of the sign of an arithmetic expression), the arithmetic operations are initially applied to the intervals. After each arithmetic operation the result (an interval) is adjusted to guarantee that it will still contain the exact result of the operation (this is called the containment property). At the end, if the sign of the exact result can be safely inferred based on the sign of the bounds of the interval, its 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.

The key to the correct and efficient implementation of operations with interval arithmetic is the fact that the IEEE-754 standard for floating-point numbers explicitly define how the 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" [27]. IEEE-754 also defines three rounding-modes (that can be selected at runtime): the results of the operations can be rounded to the nearest representable floating-point value, towards $-\infty$ or $+\infty$ (which selects, respectively, the previous or the next nearest representable floating-point numbers).

These rounding modes are employed to adjust the intervals after each arithmetic operation, which guarantees that they always contain the exact value of the expressions. [27] illustrates this process with the addition operation. 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 $xExact + yExact$.

Since the intervals are computed in a way that the containment property is always preserved, if both bounds have the same sign then this sign is equal to the exact sign of the expression. Otherwise, the interval cannot be employed to infer the exact sign and thus, the expression will have to be re-evaluated with exact arithmetic (we refer to this as an interval failure). For example, if $xExact$ is in the interval $[0.01, 0.03]$, then $xExact$ is certainly a positive number. However, if $xExact$ is in the interval $[-0.0001, 0.0001]$, then the sign of $xExact$ can be either negative, zero or positive.

Since the roundoff errors accumulate, the width of the intervals increases as arithmetic operations are performed and thus, the deeper the computation tree is, the higher are the chances that computation with exact arithmetic will be necessary, which could slow down the algorithms. However, many practical algorithms do not present this problem [27].

While arithmetic filters can accelerate predicates, in some situations the exact computation cannot be avoided. For example, exact arithmetic would be necessary in operations where new geometric objects (e.g.: points) have to be computed (these types of operations are called *geometric constructions*). To illustrate this example, consider the problem of computing pairwise intersections of line segments: arithmetic filters could be employed to accelerate the orientation predicates employed to detect if two line segments do intersect, but exact arithmetic is necessary in order to output the (exact) coordinates of the vertices generated by the intersection of pairs of edges.

The excellent Computational Geometry Algorithms Library (CGAL) [35] supports exact computation through the use of arbitrary precision rational numbers (it also supports other number types) and arithmetic filters in its algorithms. Furthermore, this library provides a framework that allows programmers to easily develop algorithms with arithmetic filters.

There are multiple types of arithmetic filters [27]. Listing 1 illustrates one of the ways to develop an arithmetic filter using C++ and CGAL: variables with the suffix *_exact* were created as GMP[13] (*GNU Multiple Precision Arithmetic Library*) arbitrary precision rationals (which are represented using the *mpq_class* type) while the ones with suffix *_interval* were defined using the interval arithmetic number type provided by CGAL. Arithmetic and boolean operators are overloaded for both the interval and arbitrary precision arithmetic types. If the comparison (line 8) cannot be evaluated safely, CGAL throws an *unsafe_comparison* exception. Once that exception is caught, the predicate can be re-evaluated using the exact version of the respective variables (line 14).

A challenge happens when a sequence of operations needs to be performed: in this situation, 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 DAG (directed acyclic graph) to represent the history of operations employed to generate each geometric object.

This kind of filter is transparent to the user (not requiring an explicit *try... catch* block similar to the one shown above). For example, if the test

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 }
```

if$(a + 2 * b + c < 0)$ is performed, then intervals will be employed to try to evaluate the test without the necessity of computing with the rationals. Assume $temp = a + 2 * b + c$ is the temporary value computed during the evaluation of *if*$(a + 2 * b + c < 0)$. If the sign of *temp* cannot be safely evaluated, its precision is increased (for example, by recomputing its value using rationals). This can be performed because the DAG associated to *temp* represents the history of operations that originated that value. I.e., *temp* knows it was computed by multiplying b by 2 and adding the result to a and c. This exact re-evaluation is lazily delayed until it is really needed ("as hopefully it won't be needed at all" [27]).

While these filters have some advantages (for example, they are efficient and can be easily and transparently used by developers), they also have some drawbacks. For example, the history DAG has a significantly high memory consumption, is hard to be maintained and is not thread-safe. Thus, even operations that do not modify the geometric objects (for example, "read-only" operations such as orientation predicates) often cannot be executed in

parallel [19].

2.3. High-performance computing and CUDA

The advent of powerful multi-core CPUs and General Purpose GPUs (*GPGPUs*) with thousand of cores has increased the computing capability of relatively inexpensive computers. For example, currently (2020) a NVIDIA GeForce 1070 Ti (a GPU with 2432 cores) can be purchased for $700 USD and provide 8 Tflop/s of peak floating-point performance. Thus, it is important to design parallel algorithms able to use this computing power.

High-performance computing has been employed to accelerate some geometric algorithms. For example, Geometric Performance Primitives (GPP), the commercial product described in [2], performs (non-exact) map overlays using GPUs.

Zhou et al. [19] and Magalhães et al. [25] have developed parallel (for shared-memory multi-core CPUs) and exact algorithms for performing boolean operations on 3D meshes. Zhou et al. [19] uses 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. [25], on the other hand, achieved thread-safeness 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 is required. The algorithms employed in arbitrary-precision arithmetic "are not easily portable to highly parallel architectures, such as GPUs or Xeon Phi" [28]. One of the reasons for this is the typically non-trivial memory management required by this kind of computation [20].

Thus, libraries for performing higher-precision arithmetic on GPUs (such as CAMPARI [20] 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 [5]. 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) these predicates that failed on the GPU.

While there has been research [6, 7] on the field of 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 implementations are in accordance with the standard since compute capabilities 1.3 and 2.0, respectively [37]. 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 [6].

In order to make interval arithmetic transparent during the evaluation of geometric predicates, we created a separate class, based on Collange et al. [6], to perform the calculations. Through operator overloading, the predicate code remains clean and concise, once the compiler intrinsics are hidden from the user.

For example, as mentioned by Collange et al. [6], the addition of two intervals $[a, b]$ and $[c, d]$ can be performed using the expression $[a, b] + [c, d] = [\underline{a + c}, \overline{b + d}]$ (where $\underline{a + c}$ and $\overline{b + d}$ indicate, respectively, the expression is rounded towards $-\infty$ and $+\infty$). Listing 2 illustrates the implementation of the addition method, where the CUDA C functions *__dadd_rd* and *__dadd_ru* switches the double precision floating point rounding mode for additions to $-\infty$ and $+\infty$, respectively.

Besides the other arithmetic operators, whose implementations are similar to addition, 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.

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 };
```

If the sign can't be inferred from the interval's bounds a special error flag is returned instead. The 2D orientation predicate, described in Section 2.2, can be easily implemented on the GPU side with interval arithmetic using our class, as shows 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 [5], and reinforced by our case study (sections 4 and 5) interval failures are rare and they usually do not affect the algorithms' overall performance.

Since GPUs are SIMT (*Single Instruction, Multiple Threads*) devices, its processing power can be explored by applying the same operation (for example, evaluating orientation predicates) on multiples triples of points in batch.

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

To evaluate the ideas presented in this paper, we have implemented a fast and exact algorithm for detecting red-blue intersection of triangles in 3D. Given two sets of triangles T_1 and T_2 (assume the red and blue triangles are from, respectively, T_1 and T_2), the objective is to find the pairs composed of red and blue triangles that do intersect.

The main idea of our implementation is to index the triangles using a uniform grid. Then, the pairs of triangles that potentially do intersect (according to the index) are culled using a fast bounding-box test. The pairs that may still intersect are, then, evaluated using a triangle-triangle algorithm (which employs the geometric predicates with interval arithmetic described in this

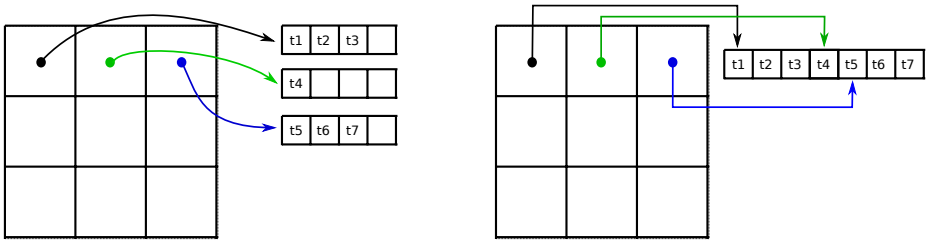


Figure 2: Dynamic array versus ragged array - 3×3 uniform grid using dynamic arrays (a) versus ragged array (b). Only the memory related to the first row of the grid is shown. Source: [24]

paper). Finally, the unreliable results (which happened due to interval arithmetic failures) 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, 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 [24]. If the input is uniformly independent and identically distributed, the expected size of the resulting subset is linear on the size of the input plus the output [1, 11, 18].

Given the sets of triangles T_1 and T_2, a grid G with resolution r (thus, containing $r \times r \times r$ cells) and dimensions equal to the bounding-box containing both T_1 and T_2 is created. Then, for each triangle t from the two input sets, t is inserted into the grid cells its bounding-box intersects. The intersecting triangles can be found by, for each grid cell c, testing all the pairs of red and blue triangles from c for intersection.

Similarly to Magalhães et al. [25], we have chosen to use a ragged array as the underlying data structure to implement the uniform grid. The ragged array stores a collection of arrays in a contiguous block of memory, by keeping track of each array's initial position. This has the advantage of being more cache friendly than storing one dynamic array per cell, since it can represent the entire grid in contiguous memory [24]. The entire structure being in contiguous memory also makes it easier to transfer the grid to the GPU. Figure 2 illustrates the indexing of a mesh using a 2D uniform grid built with a dynamic array (a), and a ragged array (b).

4.2. Bounding-box Culling

In big input meshes, the number of pairs of triangles to test for intersection may be large, even after indexing them with a uniform grid. Since the intersection between two bounding-boxes can be exactly determined using floating-point arithmetic (they consists in performing comparison operations with their bounds), testing them for intersection is faster than testing a pair of triangles. Thus, a culling is performed in order to quickly eliminate pairs of triangles that do not intersect according to their bounding-boxes.

Even though testing a pair of bounding-box using floating-point arithmetic is an exact operation, to ensure correctness it is important that the box B_T of each triangle T completely contains it. For example, if the input data is represented as double-precision floating-point numbers, but the bounds of the boxes are in single-precision: if x_d is the largest x coordinate of T, when x_d is converted to its single-precision representation x_f, x_f may be smaller than x_d (then, the box would not completely contain T). A careful implementation should change the floating-point rounding mode in order to round up in this conversion (and round down the smaller coordinates).

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, and if their bounding-box overlap, the thread increments a counter (stored in the GPU global memory) via an *atomicAdd* operation (available in CUDA). As it will be shown in the experiments, the synchronized operation does not impose a significant performance overhead. If this step was a bottleneck of the algorithm, it could be improved by employing a more sophisticated reduction operation using a faster memory from the hierarchy.

The counter is returned to the CPU which allocates an array to hold all the pairs whose bounding-boxes do overlap and, consequently, 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 increment it. This ensures each thread will insert a triangle into a different position of the array.

To perform these passes, the pairs each thread will be processing 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 [26]. 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 [26] we presented three techniques for delegating work to be done by the GPU threads when there is a large number of 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 problem from this case study.

In this paper we will employ the method which presented 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 [26]).

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, which 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 triangle 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 consists in the triangles with indices $P\%T_{2c}[C]$ (in mesh T_2) and d $P/T_2c[C]$ (in mesh T_1).

The following examples (adapted from [26]) illustrate how threads are associated to the triangles they are processed:

Table 1 shows the distribution of triangles from two meshes in a uniform with 4 cells. For example, cell 0 has 3 triangles from mesh T_{1c} and 4 triangles

from cell T_{2c} and, thus, 12 pairs of triangles will have to be evaluated for intersection in that cell.

If the algorithm is configured such that each CUDA block will have 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).

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

Table 1: Uniform grid cells. Adapted from: [26]

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

Table 2: GPU blocks of threads. Adapted from: [26]

This strategy balances the amount of work performed by each thread block, since all blocks are configured to process the same number of pairs of triangles (except for the last block processing a cell, which may evaluate fewer pairs).

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

Other strategies evaluated in [26], on the other hand, presented challenges such as load unbalance. 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 (which is already located on the GPU memory) of potentially intersecting pairs of triangles. These pairs are, then, tested for intersection.

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

After the arrays are allocated, a kernel is launched on the GPU in order to evaluate the pairs for intersection. Each thread evaluates the intersection of a pair of triangles (t_1,t_2), testing its intersection by employing five 3D orientation predicates in order to verify if any edge of t_1 intersects t_2 (and vice-versa), as described in [29].

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. Thus, thread-safe *atomicAdd* operations are performed in order to ensure correctness.

Since one triangle may be inserted into two different uniform grid cells (since it may not be completely inside a cell), pairs of triangles may be tested for intersection more than once and, therefore, the resulting array of intersections may contain duplicates. These duplicates are removed in the next step of the algorithm.

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 by employing a parallel radix-sort algorithm and, then, the duplicates are removed. These two operations are performed with the *sort* and *unique* functions provided by the *Thrust* library [16].

These two arrays are, then, copied to the CPU memory. If the intersections were not needed on the CPU (for example, if the intersection computation was part of a bigger algorithm running on the GPU), this data transference would not be necessary (only the array with interval failures would need to be copied to the CPU).

The next step is to traverse the array with the pairs which resulted in interval failures and re-evaluate them using arbitrary precision rational numbers. Since the rationals are implemented on the CPU, the pairs are evaluated in parallel using OpenMP.

If the intervals are implemented using single-precision floating-point numbers, the post-processing time could 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 [31].

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

5. Experiments

The fast algorithm for intersecting triangles was implemented on C++ and CUDA. It was evaluated on a dual Intel Xeon E5-2660 CPU at 2 GHz (3.2GHz Turbo Boost), 256 GB of RAM and a NVIDIA Quadro RTX 8000 GPU. Arbitrary-precision arithmetic was provided by the GMP library [13].

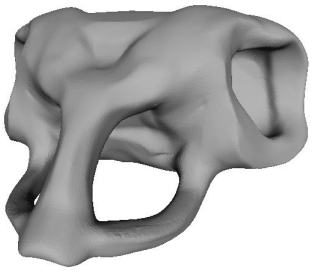
In all test cases a uniform grid with $100 \times 100 \times 100$ cells has been created. However, there are heuristics for automatically choosing a grid resolution basing on statistics about the input datasets [24, 3]. For example, the grid size could be determined as a function of the input size in a way that the expected number of pairs of edges per cell is a given constant. As shown by Magalhães et al. [24], the range of grid configurations with reasonable performance optimum is broad.

Experiments have been performed using meshes available in two public repositories. The Armadillo mesh was downloaded from the Stanford repository [36] and the other ones were downloaded from Thingi10K [39]. These meshes have been tetrahedralized using GMSH [12] before being employed in the experiments. Table 3 describes the dataset and Figure 3 illustrates it. As it can be seen, the size of these datasets ranged from 600 thousand to 8 million triangles. Figure 4 shows the interior of mesh 914686 after tethraderalization, and Figure 5 shows two overlaid pairs of meshes employed in the experiments.

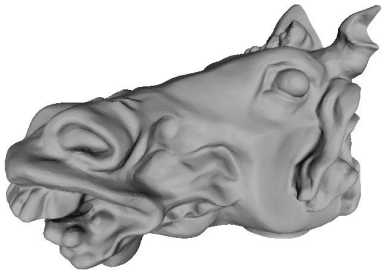
Differently from CPUs, GPUs typically have significantly more single-precision floating point units than double-precision (since they are mainly focused on applications where the lower precision of floats is acceptable) [33]. For example, the RTX 8000 GPU employed in these experiments have a peak single and double-precision performance of, respectively, 16 TFLOPs and 0.5 TFLOPs. Thus, we have also implemented a version of the GPU code employing intervals with single-precision floats in order to better utilize this computing power. As described in section 4.5, the arithmetic failures are firstly re-evaluated with double-precision intervals in the post-processing step and only predicates failing also in this second evaluation are processed with rationals.

(a) Armadillo

(b) 914686

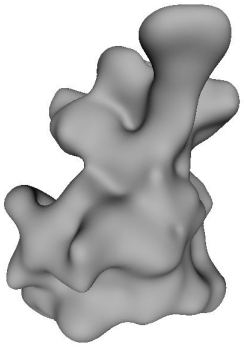


(c) 260537



(d) 68380

(e) 518092



(f) 461112

Figure 3: Meshes employed in the experiments.

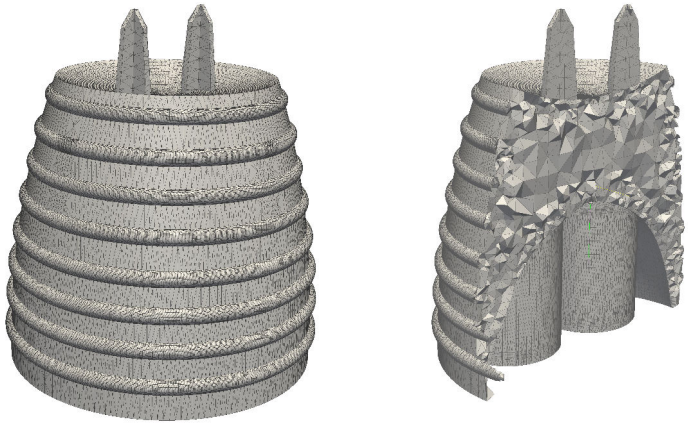
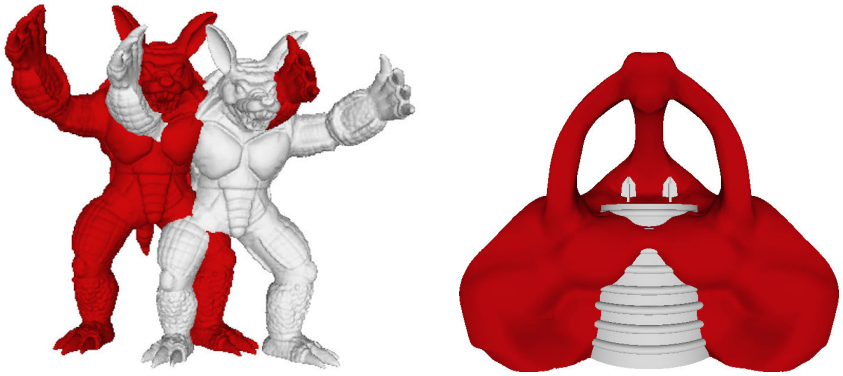


Figure 4: Mesh 914686 (left). The right figure illustrates its tetrahedralized interior (Source: [24]).



(a) (b)

Figure 5: Examples of pairs selected for intersection tests. (a) Armadillo (red) and ArmadilloTranslated (white) (b) 260537 (red) and 914686 (white).

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

Table 3: Number of vertices and triangles in the meshes employed in the experiments.

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 [34]). This is the baseline implementation.
- *GPUDouble*: parallel GPU implementation using double-precision intervals implemented as described in this paper.
- *GPUFloat*: the same implementation as GPU double, but employs single-precision floats in the intervals.

5.1. Experiments on the RTX 8000 GPU

Tables 4 and 5 present the results obtained during the intersection of triangles from pairs of the input meshes. We assume the uniform grid index is already created and loaded in the CPU memory.

The pre-processing time includes accessing the index and performing bouding-box tests in order to cull the number of triangles actually tested for intersection. Considering the GPUs version, this time also includes processing the uniform grid in order to distribute the work to the threads (as described in section 4.3).

The intersection time only includes evaluating pairs of triangles for intersection using interval arithmetic.

Post-processing time includes sorting the pairs of intersecting triangles in order to remove duplicates and re-evaluating (on the CPU) the computations whose intervals failed.

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

Finally, the last three rows of each table includes the numbers of, respectively, bounding-box tests performed by the algorithm, intersection tests (i.e., this is the number of pairs of triangles whose bounding-boxes do intersect) and actual number of intersections reported by these tests.

Considering the *CPU* version of the algorithm, the bottleneck in all test cases was testing pairs of triangles for intersection with the intervals. 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 floating-point computing power of the RTX 8000 GPU.

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

In all test-cases with *GPUFloat*, the bottleneck was the memory transference operations. More than 80% of the total memory transfer time was used for sending the uniform grid, the triangles and the auxiliary arrays used for the implicit thread association to the GPU, in the beginning of the algorithm. Indeed, if memory operations were not considering during the evaluation, the best total speedup would have been 731× instead of 414×. Thus, applications not requiring frequent data transference between the CPU and GPU could benefit even further from our method.

The worst total speedup of *GPUFloat* was 127× happened 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% (which was associated to the best total speedup) in the other tests cases. As a result, in this dataset 21% of the pairs of triangles selected at the indexing step reaches 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.

Concerning the two GPU implementations, their performance was similar considering the memory transference and pre-processing steps (in the worst case *GPUDouble* was 10% slower than *GPUFloat* in these steps). 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.

Dataset	260537 vs 914686				
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
Memory transference	-	0.14	-	0.14	-
Total time	57.62	1.29	45	0.25	232
#bounding-box tests			109.1×10^6		
#intersection tests			59.5×10^6		
#intersections			14.0×10^6		

Dataset	68380 vs 914686				
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
Memory transference	-	0.25	-	0.24	-
Total time (s)	84.71	2.00	42	0.41	208
#bounding-box tests			152.1×10^6		
#intersection tests			88.6×10^6		
#intersections			21.2×10^6		

Table 4: Times (in seconds) spent by the the intersection tests between meshes 260537 vs 914686 and 68380 vs 914686. Columns 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
Memory transference	-	0.32	-	0.33	-
Total time (s)	71.03	1.86	38	0.56	127
#bounding-box tests			339.0×10^6		
#intersection tests			72.8×10^6		
#intersections			16.8×10^6		

Dataset	461112 vs 518092				
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
Memory transference	-	0.56	-	0.51	-
Total time (s)	482.99	8.42	57	1.17	414
#bounding-box tests			655.9×10^6		
#intersection tests			505.4×10^6		
#intersections			17.8×10^6		

Table 5: Times (in seconds) spent by the the intersection tests between meshes Armadillo vs ArmadilloTranslated and 461112 vs 518092. Columns Speedup shows the speedup of the corresponding GPU version of the algorithm when compared against the CPU one.

Dataset	260537 x 914686					
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
Memory transf.	-	-	0.14	0.14	0.14	0.14
Total time	57.62	102.54	1.29	2.22	0.25	0.26

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

Considering the experiments where meshes 461112 and 518092 were intersected, for example, the post-processing time of $0.05s$ for *GPUFloat* was composed of $0.011s$ for sorting the resulting intersections and removing duplicates, $0.003s$ for filtering the pairs of triangles whose intersection computations had interval failures and $0.039s$ 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 failure and, thus, only spent time sorting the results to remove duplicates.

Considering all the experiments, in the worst case (intersection of meshes 68380 and 914686), 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 it can be seen, detecting intersection 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 a NVIDIA 1070 Ti GPU (8 GB of RAM), AMD Ryzen 5 1600 CPU 3.2 GHz (12 hyperthreads), $16GB$ of RAM in order to evaluate the proposed technique on

Dataset	68380 x 914686				
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
Memory transf.	-	0.19	-	0.19	-
Total	84.86	2.94	29	0.54	158

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

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 $0.05s$ to $0.20s$ when compared against the time on the RTX 8000 GPU. Because of the data transference operations and sequential steps of the algorithm, the difference in the total running-time (for *GPUFloat*) is smaller (the total time increased from $0.41s$ to $0.54s$ 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).

The obtained performance and exactness makes this technique applicable for interactive applications (particularly on the fields of CAD, GIS, computational geometry, and 3D modeling).

As future work, 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, a significant overhead is related to the communication between the CPU and the GPU. Reducing this communication (e.g., by moving the combinatorial part of the algorithms to the GPU) could lead to a performance improvement.

Finally, testing this technique in other architectures is also a future work: 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).

7. Acknowledgement

This research was partially supported by CAPES and FUNARBE.

References

[1] V. Akman, Wm. Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. 1989. Geometric Computing and the Uniform Grid Data Technique. *Comput. Aided Des.* 21, 7 (Sept. 1989), 410–420.

[2] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. Robust and Efficient Polygon Overlay on Parallel Stream

Processors. In *Proc. 21st ACM SIGSPATIAL Int. Conf. Advances Geographic Information Systems (SIGSPATIAL'13)*. ACM, New York, NY, USA, 304–313. https://doi.org/10.1145/2525314.2525352

[3] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. 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, 304–313.

[4] Alberto Belussi, Sara Migliorini, Mauro Negri, and Giuseppe Pelagatti. 2016. Snap Rounding with Restore: An Algorithm for Producing Robust Geometric Datasets. *ACM Trans. Spatial Algorithms and Syst.* 2, 1, Article 1 (March 2016), 36 pages. https://doi.org/10.1145/2811256

[5] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. 2001. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics* 109, 1-2 (2001), 25–47.

[6] Sylvain Collange, Marc Daumas, and David Defour. 2012. Chapter 9 - Interval Arithmetic in CUDA. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 99 – 107. https://doi.org/10.1016/B978-0-12-385963-1.00009-5

[7] Sylvain Collange, Jorge Flórez, and David Defour. 2008. A GPU interval library based on Boost.Interval. In *8th Conference on Real Numbers and Computers*. 61–71.

[8] Mark de Berg, Dan Halperin, and Mark Overmars. 2007. An intersection-sensitive algorithm for snap rounding. *Computational Geometry* 36, 3 (Apr. 2007), 159–165.

[9] Marcelo de Matos Menezes, Salles Viana Gomes Magalhães, W. Randolph Franklin, Matheus Aguilar de Oliveira, and Rodrigo E. O. Bauer Chichorro. 2019. Accelerating the exact evaluation of geometric predicates with GPUs. In *28th International Meshing Roundtable*, Suzanne Shontz, Joaquim Peiro, and Ryan Viertel (Eds.). Buffalo, NY, USA. https://doi.org/10.5281/zenodo.3653101

[10] European Space Agency. 2015. Ariane 501 inquiry board report. (2015). ravel.esrin.esa.it/docs/esa-x-1819eng.pdf (Retrieved on 06/15/2015).

[11] Wm. Randolph Franklin, Narayanaswami Chandrasekhar, Mohan Kankanhalli, Manoj Seshan, and Varol Akman. 1988. Efficiency of uniform grids for intersection detection on serial and parallel machines. In *New Trends in Computer Graphics (Proc. Computer Graphics Int.'88)*, Nadia Magnenat-Thalmann and D. Thalmann (Eds.). Springer-Verlag, Berlin, Germany, 288–297.

[12] Christophe Geuzaine and Jean-François Remacle. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *Int. J. for Numerical Methods in Eng.* 79, 11 (May 2009), 1309–1331. https://doi.org/10.1002/nme.2579

[13] Torbjorn Granlund and the GMP development team. 2014. *GNU MP: The GNU Multiple Precision Arithmetic Library* (6.0.0 ed.). http://gmplib.org/ (Retrieved on 10/19/2017).

[14] John Hershberger. 2013. Stable snap rounding. *Comput. Geom.* 46, 4 (May 2013), 403–416.

[15] John D Hobby. 1999. Practical segment intersection with finite precision output. *Comput. Geom.* 13, 4 (Oct. 1999), 199–214.

[16] Jared Hoberock and Nathan Bell. 2010. *Thrust: A Parallel Template Library*. Version 1.7.0.

[17] Christoff M. Hoffman. 1989. The Problems of Accuracy and Robustness in Geometric Computation. *Comput.* 22, 3 (Mar. 1989), 31–40.

[18] Sara Hopkins and Richard G. Healey. 1990. A Parallel Implementation of Franklin's Uniform Grid Technique for Line Intersection Detection on a Large Transputer Array. In *4th Int. Symp. Spatial Data Handling*, Kurt Brassel and H. Kishimoto (Eds.). Zürich, 95–104.

[19] Alec Jacobson, Daniele Panozzo, et al. 2016. *libigl: A Simple C++ Geometry Processing Library*. http://libigl.github.io/libigl/ (Retrieved on 10/18/2017).

[20] Mioara Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. 2016. CAMPARY: CUDA multiple precision arithmetic library and applications. In *International Congress on Mathematical Software*. Springer, 232–240.

[21] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Keng Yap. 2008. Classroom Examples of Robustness Problems in Geometric Computations. *Comput. Geom.* 40, 1 (May 2008), 61–78. https://doi.org/10.1016/j.comgeo.2007.06.003

[22] Chen Li, Sylvain Pion, and Chee Keng Yap. 2005. Recent progress in exact geometric computation. *The J. Log. Algebr. Program.* 64, 1 (July 2005), 85–111. https://doi.org/10.1016/j.jlap.2004.07.006

[23] Mian Lu, Bingsheng He, and Qiong Luo. 2010. Supporting extended precision on graphics processors. In *Proceedings of the sixth international workshop on data management on new hardware*. ACM, 19–26.

[24] Salles V. G. Magalhães and W. Randolph Franklin. 2017. *Exact and parallel intersection of 3d triangular meshes*. Ph.D. Dissertation. Rensselaer Polytechnic Institute, USA.

[25] Salles V. G. Magalhães, W Randolph Franklin, and Marcus VA Andrade. 2017. 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, 44.

[26] Marcelo Menezes, Salles V. G. Magalhães, Matheus Aguilar, W. Randolph Franklin, and Bruno Coelho. 2020. Employing GPUs to accelerate exact geometric predicates for 3D geospatial processing. In *2nd ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2020)*, John Krumm (Ed.). ACM. https://www.spatialgems.net/

[27] Sylvain Pion and Andreas Fabri. 2011. A generic lazy evaluation scheme for exact geometric computations. *Sci. Comput. Program.* 76, 4 (Apr. 2011), 307 – 323.

[28] Valentina Popescu. 2017. *Towards fast and certified multiple-precision librairies*. Ph.D. Dissertation. Université de Lyon.

[29] Rafael Jesús Segura and Francisco R. Feito. 2001. Algorithms to Test Ray-Triangle Intersection. Comparative Study. In *The 9-th Int. Conf. Central Europe Comput. Graph., Visualization Comput. Vision'2001, WSCG 2001*. 76–81.

[30] Jonathan Richard Shewchuk. 1997. Adaptive Precision FloatingPoint Arithmetic and Fast Robust Geometric Predicates. *Discret. & Comput. Geom.* 18, 3 (Oct. 1997), 305–363.

[31] J. R. Shewchuk. 1997. Adaptive Precision FloatingPoint Arithmetic and Fast Robust Geometric Predicates. *Discret. & Comput. Geom.* 18, 3 (Oct. 1997), 305–363.

[32] Robert Skeel. 1992. Roundoff error and the Patriot missile. *SIAM News* 25, 4 (July 1992), 11.

[33] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. 2019. Summarizing CPU and GPU Design Trends with Product Data. *arXiv preprint arXiv:1911.11313* (2019).

[34] The CGAL Project. 2015. CGAL, *Computational Geometry Algorithms Library*. http://www.cgal.org (Retrieved on 10/19/2017).

[35] The CGAL Project. 2016. *CGAL User and Reference Manual* (4.8 ed.). http://doc.cgal.org/4.8/Manual/packages.html (Retrieved on 10/19/2017).

[36] The Stanford 3D Scanning Repository. 2016. "The stanford 3D scanning repository". (2016). http://graphics.stanford.edu/data/3Dscanrep/ (Retrieved on 10/19/2017).

[37] Nathan Whitehead and Alex Fit-Florea. 2011. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. *rn (A+ B)* 21, 1 (2011), 18749–19424.

[38] Chee Keng Yap. 1997. Towards exact geometric computation. *Comput. Geom.* 7, 1–2 (Jan. 1997), 3 – 23.

[39] Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).