MARCELO DE MATOS MENEZES

FAST AND EXACT EVALUATION OF GEOMETRIC PREDICATES USING GRAPHICS PROCESSING UNITS

Dissertation submitted to the Computer Science Graduate Program of Universidade Federal de Viçosa in partial fulfillment of the requirements for the degree of *Magister Scientiae*.

Advisor: Salles Viana Gomes de Magalhães

VIÇOSA - MINAS GERAIS

2021

Ficha catalográfica elaborada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa

T
M543f
2021

Menezes, Marcelo de Matos, 1993-
Fast and exact evaluation of geometric predicates using graphics processing units / Marcelo de Matos Menezes. – Viçosa, MG, 2021.
63 f. : il. (algumas color.) ; 29 cm.

Orientador: Salles Viana Gomes de Magalhães.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 61-63.

1. Unidades de processamento gráfico.
2. Processamento paralelo (Computadores). 3. Geometria - Processamento de dados. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 006.663

MARCELO DE MATOS MENEZES

FAST AND EXACT EVALUATION OF GEOMETRIC PREDICATES USING GRAPHICS PROCESSING UNITS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 10 de fevereiro de 2021.

Assentimento:

Marcelo de Matos Menezes
Autor

Salles Viana Gomes de Magalhaes
Orientador

Acknowledgment

I would like to thank my advisor, Dr. Salles Magalhães, for his support and guidance during my studies. His dedication, enthusiasm, and knowledge were fundamental for the accomplishments of this work.

To Dr. W. Randolph Franklin, for the essential inputs and corrections on our papers, and for representing us in international conferences. Also, to the other members of our research group, Matheus, Rodrigo, and Bruno.

To the professors of UFV's Departamento de Informática. In special, Drs. Ricardo Ferreira and Carlos Goulart, for the corrections and insightful suggestions to improve this dissertation, and Drs. Marcus Andrade and André dos Santos, who also contributed to my professional and intellectual growth.

To Luciana, for all the love, companionship, and encouragement.

To my parents Dirlene and Fernandes, my siblings Flávio and Driele, and my uncle Ildeu, for always supporting and believing in me. A special thanks to my late grandmother, Hélia. Without her efforts, I could not have followed my dreams.

Finally, I would like to express my appreciation to the friends I've made during my years at Universidade Federal de Viçosa. In particular, to Felipe and Gabriel, for all the moments of laughter.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

Resumo

MENEZES, Marcelo de Matos, M.Sc., Universidade Federal de Viçosa, fevereiro de 2021. **Avaliação rápida e exata de predicados geométricos utilizando unidades de processamento gráfico**. Orientador: Salles Viana Gomes de Magalhães.

A precisão finita da aritmética de ponto flutuante é um grande desafio na área de geometria computacional, uma vez que algorimos desenvolvidos cuidadosamente podem falhar devido a erros de arredondamento. Embora existam pesquisas voltadas ao desenvolvimento de algoritmos geométricos exatos, as técnicas existentes até então ainda podem ser melhoradas em termos de desempenho. Neste trabalho é proposto um método eficiente e exato para avaliação de predicados geométricos utilizando a unidade de processamento gráfico, no qual, através de uma implementação de filtragem de aritmética na GPU, a maior parte dos predicados podem ser calculados sem erros de arredondamento. Os (poucos) resultados que não são confiáveis, são reavaliados usando números racionais de precisão arbitrária em paralelo na CPU. A eficiência do método foi medida com implementações de algoritmos geométricos em 3 estudos de caso: interseção de segmentos em 2D, interseção de segmento e triângulo em 3D e interseção de triângulos em 3D. Em cada passo, novas técnicas foram propostas para acelerar os gargalos de desempenho. Nos últimos experimentos, uma comparação do método contra uma implementação sequencial na CPU resultou em aceleração (*speedups*) de até 1936× para avaliação de interseções, e 414× considerando o tempo total do algoritmo. Tal eficiência mostra que a técnica descrita nesse trabalho é ideal para acelerar a computação exata de predicados geométricos, o que deve beneficiar áreas como CAD, SIG e modelagem 3D.

Palavras-chave: Geometria computacional. Computação exata. Filtragem de aritmética. GPGPU.

Abstract

MENEZES, Marcelo de Matos, M.Sc., Universidade Federal de Viçosa, February, 2021. **Fast and exact evaluation of geometric predicates using graphics processing units**. Adviser: Salles Viana Gomes de Magalhães.

Floating point arithmetic's finite precision presents a major challenge in the field of computational geometry, since even carefully designed algorithms may fail due to round-off errors. While there has been research to develop exact geometric algorithms, the existing techniques so far can still be improved performance-wise. In this work we propose a fast and exact method for efficient geometric predicate evaluation using the graphics processing unit, in which, by implementing arithmetic filtering on the GPU, the computation of most predicates can be performed without round-off errors. The (few) results that are unreliable are reevaluated using arbitrary precision rational numbers in parallel on the CPU. We measured the efficiency of our method by implementing geometric algorithms in 3 case studies: 2D segment-segment intersection, 3D segment-triangle intersection, and 3D triangle-triangle intersection. At each step, new techniques were proposed for eliminating the current performance bottleneck. In our last experiments, a comparison of our method against a sequential CPU implementation yielded speedups of up to 1936× for intersection evaluation, and 414× for the entire running time of the algorithm. The achieved efficiency shows that our technique is ideal for accelerating exact geometric computations, and fields such as CAD, GIS, and 3D modeling should benefit from it.

Keywords: Computational geometry. Exact computation. Arithmetic filtering. GPGPU.

List of Figures

List of Tables

Contents

1 Introduction

Geometric predicates are the building blocks of many computational geometry algorithms. For example, an orientation predicate, which determines whether a point is above, below, or is collinear in relation to a line segment, can be used to detect intersections between pairs of segments. The results of such predicates can, however, be erroneously evaluated due to round-off errors caused by floating-point arithmetic. This errors can be propagated to higher level operations, which make the algorithms produce wrong results. For example, the calculated point of intersection between two line segments may not intersect either.

There are various methods that try to solve the problems caused by round-off errors. The formally proper way that guarantees robustness is to use exact computation with arbitrary precision rational numbers (LI; PION; YAP, 2005; HOFFMAN, 1989; KETTNER et al., 2008; YAP, 1997). This approach has a limitation of being too slow when compared with an inexact representation.

To improve the efficiency, while maintaining robustness, Pion and Fabri (PION; FABRI, 2011) implemented a technique called Arithmetic Filtering on the CGAL library (The CGAL Project, 2016). It uses interval arithmetic, which represents an exact number using a pair of floating points. By exploring some properties of the intervals, computation can be done using fast floating-point arithmetic and it is possible to determine if the results are reliable or not. If some results are not reliable (this is known as filter failure), they are recomputed using exact precision. Thus, the algorithms can reduce the necessity of exact computations (relying on them only in the few cases where they are really necessary).

The existing methods for solving geometric problems avoiding round-off errors can still be improved in efficiency. Oliveira et. al. (OLIVEIRA et al., 2020) studied the performance of these techniques: they employed the exact and approximate backends of PostGIS for detecting intersections between segments and triangles in 3D. In their experiments, the exact backend (implemented using CGAL's arithmetic filtering) was up to 27 times slower than the approximate one.

Since larger datasets have a higher probability of generating round-off errors, and take more time to process, the necessity of a method for geometric computation that is both exact and fast arises.

In this project we implemented a framework for fast and exact processing of geometric predicates using the GPU. The work was divided in three main steps, of which, two were published in conferences and the last one submitted to a journal. The papers

were adapted to compose chapters 2-4 of this dissertation. Since the first and third papers (chapters 2 and 4) share a common background, their beginning was extracted and composed into section 1.1 of this chapter, which introduces the core concepts of this work.

Chapter 2 explains a preliminary implementation of our framework, which was designed and tested for calculating intersections between segments in 2D. The overall idea is to send a batch of predicates to the GPU, which will evaluate them in parallel using interval arithmetic. The few unreliable results are re-evaluated using exact arithmetic (arbitrary precision rational numbers) in parallel on the CPU. Before sending the predicates to the GPU, the method employs a uniform grid for culling segments that are unlikely to intercept. The grid is traversed on the CPU, which creates a list of pairs of segments that needs to be tested for intersection.

This method achieved speedups of up to 289x for intersection evaluation (40x considering the algorithm's overall time) in our tests. The technique and its results were published in the 28th International Meshing Roundtable (IMR) (MENEZES et al., 2019).

Chapter 3 presents an improvement on the previous method, which is designed for calculating intersections between segments and triangles in 3D. A trivial extension did not yield a good speedup, mainly because the pre-processing step where the CPU traverses the grid to create a list of pairs became the bottleneck. We overcame this problem by indirect associating a GPU thread with a segment-triangle pair, with the help of 3 auxiliary arrays. We also employed a bounding-box filtering step to further cull unnecessary tests. This filtering improved the algorithm's performance, but the improvement on the sequential CPU version was higher (it creates thread divergence on the GPU, since threads of the same warp can stop execution after the test).

We tested the new method using a dataset from (REAL et al., 2019), which employs GPUs to accelerate 3D geospatial queries. Their proposed solution is not exact, though. Speedups of up to 17x were achieved when compared against the sequential CPU implementation. The paper (MENEZES et al., 2020) was published on the 2nd ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2020), and won the best paper award of the workshop.

We were invited by Elsevier's Computer-Aided Design Journal to extend the paper published in IMR. Chapter 4 was adapted from this extension, and presents new improvements on our technique.

The bounding-box filtering is done in a separate pass to minimize thread divergence, and provide a better speedup on the GPU. More importantly, the creation of the array of pairs is delegated to the GPU, using the indirect association of the previous work. We also changed the interval's bounds from double to single precision, since modern GPUs are optimized for single precision floating point arithmetic. This increases the number of filter failures, but our experiments showed that the number is still low, which results in

a good trade-off. The filter failures are reevaluated on the CPU using double precision, before relying on rational numbers, which reduces the impact even further.

These enhancements resulted in speedups of up to 407× for intersection evaluation, and 158× considering the overall algorithm, for evaluating intersections between pairs of triangles in 3D (these speedups were achieved using a NVDIA GeForce GTX 1070 Ti; when using a NVIDIA Quadro RTX 8000, the speedups were up to 1936× and 414×, respectively).

1.1 Background

This section introduces the necessary concepts for understanding the techniques proposed in this project. It begins with a refresher on floating point arithmetic and shows the impact of roundoff errors on computational geometry, while also giving examples of common techniques for solving the problem. This is followed by an explanation of interval arithmetic and arithmetic filters, the methods explored in this work for achieving exact and efficient computation of geometric predicates on the GPU. The section finishes with a discussion of existing methods for solving the problem at hand and a brief description of the current GPU architecture.

As the main parts of this dissertation are composed by papers produced along the project, the text contains contributions from the other authors.

1.1.1 Floating-point arithmetic

As stated by Muller et al. (MULLER et al., 2018), a floating-point format is partially characterized by four integers:

- a radix $\beta \geq 2$
- a precision $p \geq 2$
- two extremal exponents e_{min} and e_{max} such that $e_{min} < e_{max}$

A floating-point x can be expressed in such format as $x = M \times \beta^{e-p+1}$, where M is an integer called *integral significand* of the representation of x ($|M| \leq \beta^p - 1$), and e is an integer called the *exponent* of the representation of x ($e_{min} \leq e \leq e_{max}$).

Since floating point numbers are a small subset of real numbers, the result of an operation generally cannot be exactly representable in the system, so it needs to be rounded. Figure 1 shows five possible rounding choices for representing an exact value x in the floating point range. Suppose the horizontal line represents the real numbers, and

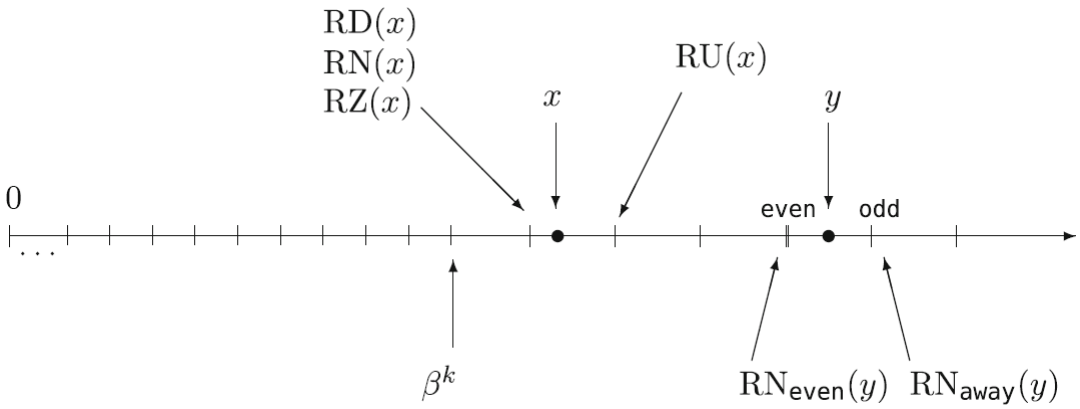


Figure 1 – Example of the effect of rounding functions for representing exact values x and y as floating point numbers. Source: (MULLER et al., 2018)

the vertical lines shows the representable floating point values. The rounding choices are defined as follows (MULLER et al., 2018):

- RD(x) (round towards $-\infty$) chooses the largest floating point value that is less than or equal to x.
- RU(x) (round towards ∞) chooses the smallest floating point value that is greater than or equal to x.
- RZ(x) (round towards zero) similar to RD(x) if x is positive, and RU(x) otherwise.
- RNeven(y) (round to nearest). If there are two nearest representable values, chooses the even one.
- RNaway(y) (round to nearest). If there are two nearest representable values, chooses the one whose magnitude is largest.

1.1.2 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 (European Space Agency, 2015) and the failure of the Patriot missile defense system (SKEEL, 1992).

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. (KETTNER et

al., 2008) 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, 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. (KETTNER et al., 2008) 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 2 shows the geometry of this experiment for $p = (0.5, 0.5)$, $u = 2^{-53}$, $q = (12, 12)$ and $r = (24, 24)$. The left image shows how the points are distributed according to the segment (p, q); the middle and right images show, respectively, the expected result, and the obtained result. As it can be seen, several points have their orientation computed incorrectly. These inconsistent results in the orientation predicates could make algorithms that use this predicate to fail (KETTNER et al., 2008).

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,

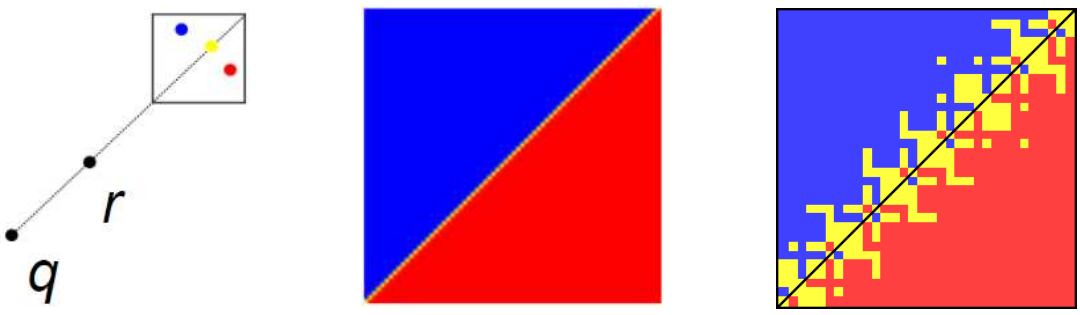


Figure 2 – Roundoff errors in the planar orientation problem. Rightmost figure source: (KETTNER et al., 2008).

nor invariant under scaling. Thus, in practice, epsilon-tweaking fails in several situations (KETTNER et al., 2008).

Snap rounding is another method to approximate arbitrary precision segments into fixed-precision numbers (HOBBY, 1999). 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 (BERG; HALPERIN; OVERMARS, 2007; HERSHBERGER, 2013; BELUSSI et al., 2016).

Shewchuk (SHEWCHUK, 1997) 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 (SHEWCHUK, 1997), 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" (SHEWCHUK, 1997).

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 (LI; PION; YAP, 2005; HOFFMAN, 1989; KETTNER et al., 2008; YAP, 1997). 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 1.1.3.

1.1.3 Arithmetic filters and interval arithmetic

One technique to accelerate algorithms based on exact arithmetic is to employ arithmetic filters and interval arithmetic (PION; FABRI, 2011). 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" (PION; FABRI, 2011). 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. (PION; FABRI, 2011) 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 (PION; FABRI, 2011).

While arithmetic filters can accelerate predicates, in some situations the exact computation cannot be avoided. For example, exact arithmetic would be necessary in

Listing 1.1 – 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 }
```

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) (The CGAL Project, 2016) 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 (PION; FABRI, 2011). Listing 1.1 illustrates one of the ways to develop an arithmetic filter using C++ and CGAL: variables with the suffix *_exact* were created as GMP (GRANLUND; the GMP development team, 2014) (*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 $if(a + 2 * b + c < 0)$ 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" (PION; FABRI, 2011)).

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 (JACOBSON; PANOZZO et al., 2016).

1.1.4 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 (2019) a NVIDIA GeForce 1080 Ti (a GPU with 3584 cores) can be purchased for $800 USD and provide 11 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 (AUDET et al., 2013), performs (non-exact) map overlays using GPUs.

Zhou et al. (JACOBSON; PANOZZO et al., 2016) and Magalhães et al. (MAGALHÃES; FRANKLIN; ANDRADE, 2017) have developed parallel (for shared-memory multi-core CPUs) and exact algorithms for performing boolean operations on 3D meshes. Zhou et al. (JACOBSON; PANOZZO et al., 2016) 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. (MAGALHÃES; FRANKLIN; ANDRADE, 2017), 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" (POPESCU, 2017). One of the reasons for this is the typically non-trivial memory management required by this kind of computation (JOLDES et al., 2016).

Thus, libraries for performing higher-precision arithmetic on GPUs (such as CAMPARI (JOLDES et al., 2016) and GARPREC (LU; HE; LUO, 2010)) 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 (BRÖNNIMANN; BURNIKEL; PION, 2001). 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 (COLLANGE; DAUMAS; DEFOUR, 2012; COLLANGE; FLÓREZ; DEFOUR, 2008) 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.

1.1.5 CUDA programming model

This subsection provides a brief explanation on the nuances of programming CUDA enabled GPUs. The interested reader should refer to the book *Professional CUDA C Programming* (CHENG; GROSSMAN; MCKERCHER, 2014) for a deeper understanding

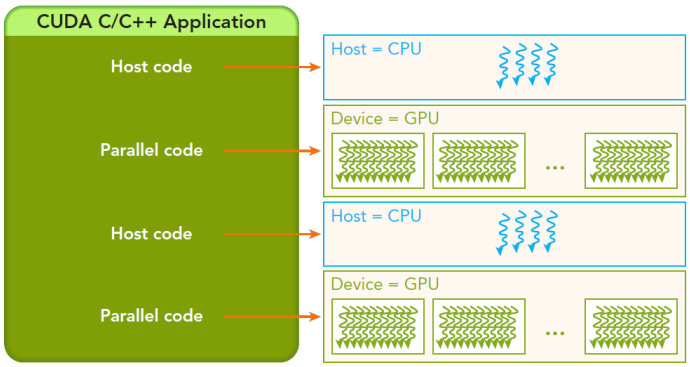


Figure 3 – The serial code (or task parallel code) is executed on the host, while the parallel part is delegated to the GPU. Source: (CHENG; GROSSMAN; MCKERCHER, 2014)

of the subject.

The CUDA programming model specifies a special type of function called kernel, where the code that runs on the GPU will reside. The CPU (host) can operate independently of the GPU (device) for most operations. Figure 3 shows the relation between host and device in a typical CUDA program.

In CUDA, the GPU threads are organized in a two-level hierarchy decomposed into blocks of threads and grids of blocks (CHENG; GROSSMAN; MCKERCHER, 2014). When the host launches a kernel, the device code is executed by the GPU threads, and the resources available per thread depends on the configuration of the hierarchy. Another important aspect of the model is how the memory system is defined. The two most important types of memories are the global (analogous to the CPU system memory) and shared (analogous to the CPU cache) memories. The shared memory can be explicitly programmed, and its availability depends on the number of threads launched per block. Figure 4 illustrates the grid and memory hierarchies.

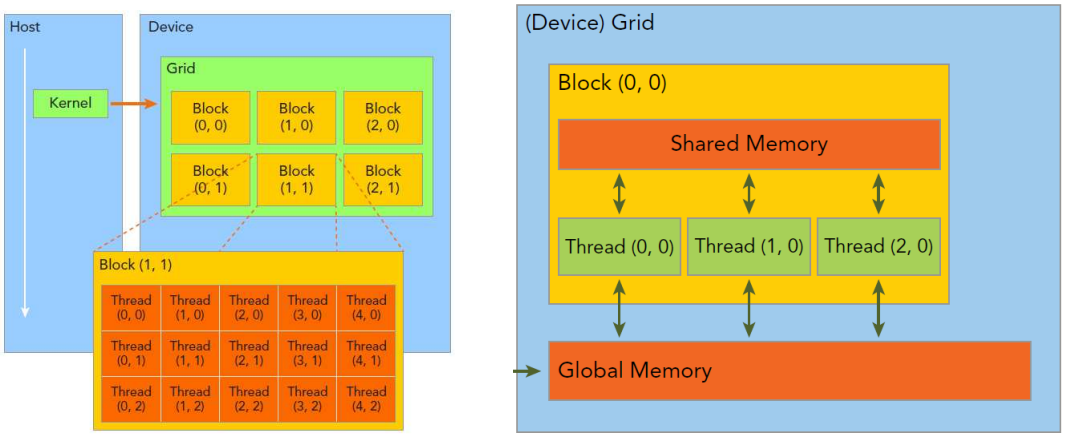


Figure 4 – CUDA's grid (left) and memory (right) hierarchies. Source: (CHENG; GROSSMAN; MCKERCHER, 2014)

A GPU is composed of multiple multiprocessors called *Streaming Multiprocessors* (SM), and each one can execute hundreds of threads concurrently. When a kernel grid is scheduled for execution, its thread blocks are distributed between the SMs depending on the number of threads and available resources. Each SM is a *Single Instruction Multiple Thread* (SIMT) device, and executes at the same time groups of 32 threads called *warps*. Each thread in a warp has its own instruction address counter and register state, and therefore, can point to different instructions. However, it is essential for the programmer to reduce the number of divergences in the kernel code, since different code paths can kill the GPU performance.

The CUDA toolkit hides much of the work needed for programming the device. The typical CUDA program follows the pattern below:

- Copy data from host to device
- Invoke kernels for operating on device data
- Copy data back from device to host

The runtime provides functions for allocating, copying and deallocating memory, such as *cudaMalloc, cudaMemcpy, cudaMemset, and cudaFree*. There is also special syntax for specifying the thread grid configuration when launching a kernel. Supposing a kernel with x blocks per grid and y threads per block, the following syntax is used for launching kernel k with arguments *arg_list*:

- k<<<x, y>>>(arg_list)

Even though the runtime has many facilities for developing GPU programs, the programmer needs to be aware of the best practices for creating optimized and reliable systems. The official toolkit documentation (NVIDIA; VINGELMANN; FITZEK, 2021) should be used as a guide when developing CUDA programs.

2 Accelerating the exact evaluation of geometric predicates with GPUs

This chapter 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 segment 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 289 times (when compared against a similar sequential implementation) in the evaluation of these predicates (and up to 40 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 VLSI design.

2.1 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 (BRÖNNIMANN; BURNIKEL; PION, 2001), 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 chapter, we propose the use of a combination of GPUs and multi-core CPUs to accelerate the evaluation of exact predicates using arithmetic filters. During the parallel evaluation of predicates, the operations with intervals are offloaded to a GPU. Then, the (few) unreliable results are filtered and re-evaluated in parallel on the CPU using multiple-precision rationals. As a result, both high-efficiency and exactness are achieved.

To obtain performance, the algorithms being accelerated should be adapted so that the geometric predicates are evaluated in batch. For example, consider the problem of computing the intersection of two triangulated meshes. One critical step consists in, given a set of pairs of potentially intersecting triangles, determine which ones do intersect. Since the intersection of two triangles can be computed using orientation predicates, this algorithm could create a list of these predicates and offload their evaluation to the GPU in batch.

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.

As a case study, we have developed a fast and exact algorithm for detecting red-blue intersections between two sets of edges in $2D$. We intend to also apply these techniques to accelerate the solution of other important problems such as performing boolean operations on polygonal maps or polyhedral meshes.

2.2 Implementing exact parallel predicates

As stated in section 1.1.3, 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 (WHITEHEAD; FIT-FLOREA, 2011). 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 (COLLANGE; DAUMAS; DEFOUR, 2012).

In order to make interval arithmetic transparent during the evaluation of geometric predicates, we created a separate class, based on Collange et al. (COLLANGE; DAUMAS; DEFOUR, 2012), 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. (COLLANGE; DAUMAS; DEFOUR, 2012), 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 2.1 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. 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 1.1.3, can be easily implemented on the GPU side with interval arithmetic using our class, as shows listing 2.2. 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 (BRÖNNIMANN; BURNIKEL; PION, 2001), and reinforced by our case study (sections 2.3 and 2.4) 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 2.1 – 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 };
```

Listing 2.2 – 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 }
```

2.3 Fast red-blue intersection tests

To evaluate the ideas presented, we have implemented a fast and exact algorithm for detecting red-blue intersection of line segments. Given two sets of segments M_1 and M_2 (assume the red and blue segments are from, respectively, M_1 and M_2), the objective is to find the pairs composed of red and blue segments that do intersect. This is performed by doing a pre-processing step with a uniform grid to cull pairs of segments that may intersect and, then, filtering the pairs that actually do intersect.

The uniform grid is typically employed in computational geometry to cull a combinatorial set of pairs of objects, generating a smaller subset containing elements that are more likely to coincide (MAGALHÃES; FRANKLIN, 2017). If the input is uniformly independently and identically distributed, the expected size of the resulting subset is linear on the size of the input plus the output (AKMAN et al., 1989; FRANKLIN et al., 1988; HOPKINS; HEALEY, 1990). Thanks to its simplicity and uniformity, it can be constructed and processed in parallel. For example, Audet et al. (AUDET et al., 2013) employed a uniform grid on a GPU parallel algorithm for map overlay and Magalhães et al. (MAGALHÃES; FRANKLIN; ANDRADE, 2017) employed it to intersect 3D meshes in parallel.

Given the sets of segments M_1 and M_2, a grid G with resolution r (thus, containing $r \times r$ cells) and dimensions equal to the bounding-box containing both M_1 and M_2 is created. Then, for each segment e from the two input sets, e is inserted into the grid cells it intersects. The intersecting segments can be found by, for each grid cell c, testing all the pairs of red and blue segments from c for intersection.

For performance and simplicity, as in Magalhães et al. (MAGALHÃES; FRANKLIN; ANDRADE, 2017), instead of rasterizing each segment s in order to determine which cells s intersects, the bounding-box b of s is computed and s is considered to intersect all grid cells intersecting b. While this may increase the number of intersection tests that will have to be performed later, the correctness of the algorithm is maintained since the grid is employed only to find a set of edges that may intersect.

Similarly to Magalhães et al. (MAGALHÃES; FRANKLIN; ANDRADE, 2017), 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. It can be easily constructed in parallel, with the cost of making two passes in the data to insert the edges, and has the advantage of being more cache friendly than storing one resizable array per cell, since it can represent the entire grid in contiguous memory (MAGALHÃES; FRANKLIN, 2017). Figure 5 illustrates these two data structures.

The creation of the ragged array storing in the grid the segments from each of the

input sets M_i ($i = 1$ or 2) is performed in two passes. First, the number of segments from M_i in each cell is counted. Then, the array is allocated (with size equal to the sum of the number of edges in all cells) and the segments are scanned again and effectively inserted into the array.

In the first pass, the bounding-box of each segment s in M_i is initially computed on the GPU. This computation is performed in parallel and basically consists in determining the grid cells containing each of the two endpoints of s (this is the only geometric operation performed during the construction of the uniform grid). Then, a counter *cellSize[c]* is created to compute the number of segments that will be inserted into each grid cell c (we refer to this as the size of the cells). Finally, each segment s is scanned (in parallel) and the counter of the cells the bounding-box of s intersects is incremented (using atomic operations).

After the cell sizes are computed, a parallel exclusive prefix-sum operation is applied to the *cellSize* array. Assume *cellStart* is the content of *cellSize* after the prefix-sum. Thus, *cellStart[0]=0*, *cellStart[1] = cellStart[0] + cellSize[0]* and, in general, *cellStart[c]= cellStart[c - 1] + cellSize[c - 1]*. Therefore, *cellStart[c]* represents the starting position of the edges of cell c in the ragged array.

In the second pass, each segment s in M_i is processed again in parallel. For each cell c intersecting the bounding-box of s, s is inserted into the position *cellStart[c]+count[c]* of the ragged array, where $count[c]$ is a counter for the current number of segments inserted into c. Since *count* may be incremented in parallel, this operation is performed using an atomic increment and capture operation (which returns the current value in count and increments it).

Once the uniform grid is constructed, a list L of the pairs of red and blue segments from all the grid cells is created. This list is generated in parallel using a strategy similar to the creation of the ragged-array: an initial pass is performed to count the number of pairs of edges in all grid cells and, then, a second one effectively inserts the pairs into the list.

The intersection between a pair of segments can be detected by evaluating 4 2D orientation predicates. Consider, for example, the segments s_1 (with endpoints A and B) and s_2 (with endpoints C and D). If the orientation of (A, B, C) has a different sign than the one of (A, B, D), then C and D are on opposite sides w.r.t. s_1 (the supporting line of s_1 intersects s_2). Similarly, if the orientation of (C, D, A) has a different sign than the one of (C, D, B), then the supporting line of s_2 intersects s_1. If both supporting lines intersect, then the segments do intersect. These 4 orientation predicates are performed in parallel on the GPU (see listing 2.2) for all pairs of segments in L.

Since edges may be inserted into multiple grid cells, a pair may be tested for intersection more than once (and, if they do intersect, multiple copies of them would be

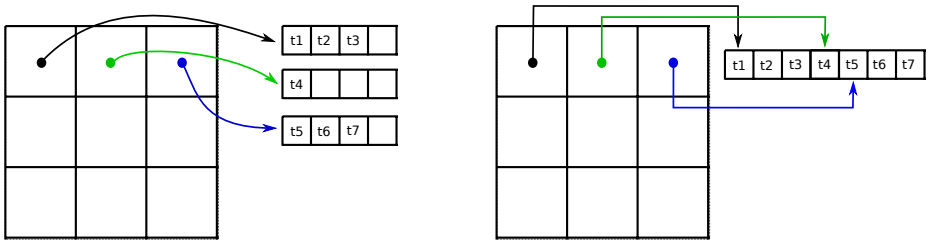


Figure 5 – Dynamic array versus ragged array - 3×3 uniform grid using dynamic arrays (a) versus ragged array (b). Only the memory related to the first row of the grid is shown. Source: (MAGALHÃES; FRANKLIN, 2017)

outputed by the algorithm). Preliminary experiments showed that a better performance is achieved when the duplicates are removed after the intersections are detected (instead of removing them before the intersection tests). This can be explained because, as it will be shown in section 2.4, detecting intersections using the GPU is a fast process.

All the geometric operations (determining the grid cells containing each segment endpoint and evaluating the orientation predicates for detecting intersections) are performed on the GPU in batch. However, since some of the operations performed with intervals (employed to determine the grid cells containing the endpoints of each segment) may return a failure code, after each batch of these operations the results are copied back to the CPU and the ones that failed are re-evaluated using arbitrary-precision rationals.

2.4 Experiments

To evaluate the proposed ideas, the fast algorithm for intersecting edges was implemented on C++ and evaluated on a AMD Ryzen 5 desktop with 6 3.2 GHz cores (and 12 hyperthreads), 16 GB of RAM and a NVIDIA GeForce GTX 1070 Ti GPU. Arbitrary-precision arithmetic was provided by the GMP library (GRANLUND; the GMP development team, 2014) and the algorithm was parallelized with OpenMP (for the code targeted to the CPU) and CUDA (for the GPU code).

In all test cases a uniform grid with $2,500 \times 2,500$ cells has been created. However, there are heuristics for automatically choosing a grid resolution based on statistics about the input datasets (MAGALHÃES; FRANKLIN, 2017; AUDET et al., 2013). 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. (MAGALHÃES; FRANKLIN, 2017), the range of grid configurations with reasonable performance optimum is broad.

Experiments have been performed using segments from four polygonal maps from two countries. The two maps from Brazil were obtained from the IBGE (the Brazilian geography agency) and represent the kinds of soil (BrSoil) and the counties (BrCounty)

	Pairs of maps evaluated					
	BrSoil	BrCounty	UsCounty	UsAquifers	UsCounty	UsCountyRot.
# of segments	211,011	326,193	3,740,989	352,924	3,740,989	3,740,989
Avg. segment length (% of bb.)	5×10^{-4}	4×10^{-4}	8×10^{-7}	1×10^{-4}	8×10^{-7}	8×10^{-7}
% of empty grid cells		86%		98%		98%
Avg. # pairs of segments/cell		0.3		2.0		34.7
# of pairs of segments		300,039		12,756,283		216,542,974
# of intersections		20,860		11,948		11,751

Table 1 – Statistics about the input datasets and about the intersection computation process.

from Brazil. The two maps from the USA were obtained from the ESRI ArcGIS and the United States National Atlas web-pages. We also performed tests intersecting the largest dataset (UsCounty) with a version of itself (UsCountyRotated) rotated by 0.1° (counter-clockwise) around the center of the bounding-box of the original map. Experiments with UsCountyRotated are particularly hard for the uniform grid because it generates a high amount of potentially intersecting pairs of edges (thus, requiring more pairs of edges for being tested for intersection).

Figure 6 illustrates four of the datasets and Table 1 present some statistics about the input maps and about the intersection computation process. As it can be seen, the size of the input datasets range from 200 thousand to 4 million segments. The average length of the segments is presented as a percentage of the diagonal of the bounding-box.

The last four rows of Table 1 present statistics about the pairs of evaluated input maps. In all cases, most of the uniform grid cells cover empty regions of the input datasets. Row *Average # pairs of segments/cell* indicates the average number of pairs of red-blue segments per non-empty cell. Row *Number of pairs of segments* indicates the total number of pairs of red-blue segments in all cells (i.e., the number of pairs tested for intersection). As it can be seen in the last row, the actual number of intersections ranged from 6% to 0.005% of the number of intersection tests performed. Indeed, the dataset which generated the largest amount of intersection tests was the one with the smallest number of actual intersections.

We compared 5 versions of the algorithm (* marks a sequential implementation):

- Rational*: sequential implementation employing only arbitrary-precision rational arithmetic. This algorithm was evaluated in order to show the benefit obtained by the arithmetic filters in the other versions.
- Interval*: same as Rational*, but employing arithmetic filters with interval arithmetic.
- Rational: parallel (CPU) version of Rational*.
- Interval: parallel (CPU) version of Interval*.

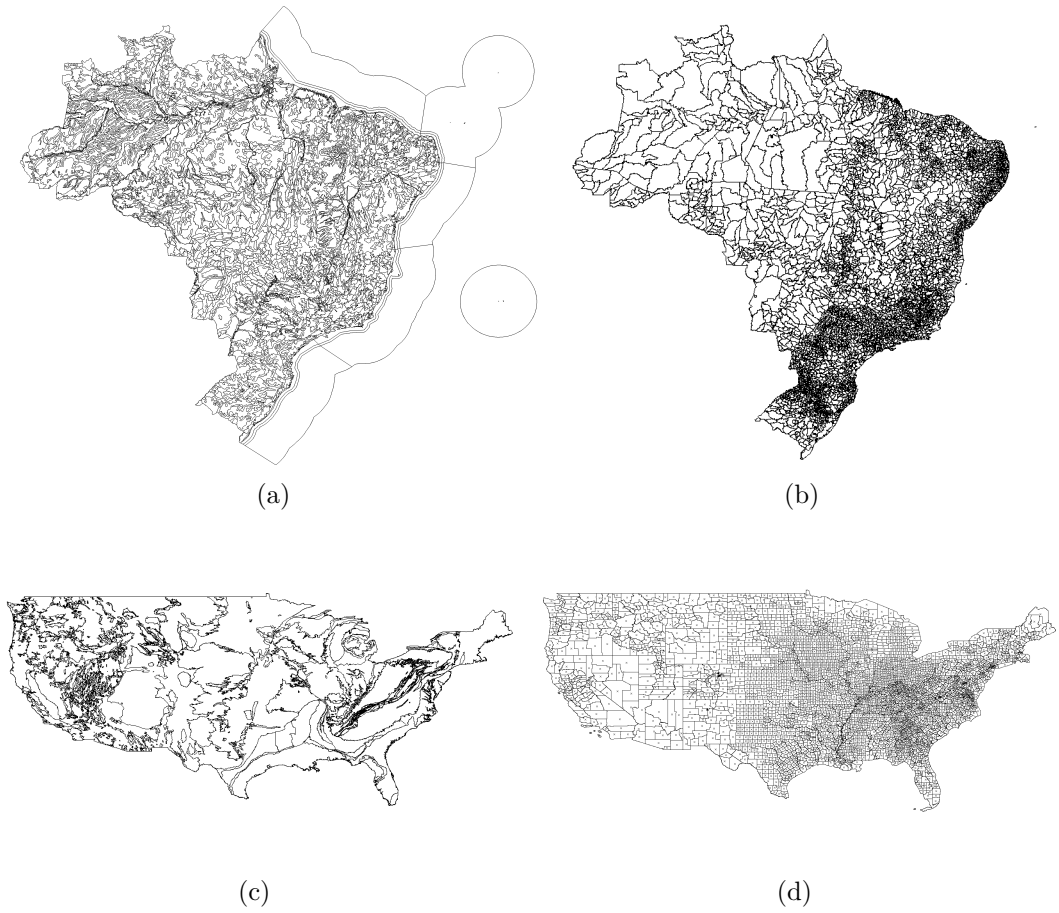


Figure 6 – Maps employed in the experiments - BrSoil (a), BrCounty (b), UsAquifers (c), UsCounty (d) (these figures are not to scale).

- GPU: parallel (using the CPU and the GPU) version of Interval*.

Furthermore, as a baseline, we also implemented an algorithm using CGAL to detect intersections. This algorithm employs CGAL's method for intersecting dD Iso-oriented Boxes as a pre-processing step to initially cull the pairs of potentially intersecting segments. This culling process is sequential and employs a hybrid method composed of a sweep-line and a streaming algorithm to detect intersection between pairs of Axis-Aligned Bounding Boxes. Then, CGAL's *do_intersect* method is employed to check if each of the remaining pairs of segment do intersect.

For exactness, the Exact_predicates_exact_constructions_kernel kernel has been employed (this CGAL kernel stores exact versions of the geometric constructors and employs arithmetic filters and lazy evaluation to accelerate the evaluation of predicates).

Table 2 presents the results obtained during the intersection of edges from pairs of input maps.

The pre-processing strategy performed by CGAL performs a better culling than the other methods, eliminating all pairs of edges whose bounding-boxes do not intersect

Datasets	BrCounty and BrSoil						
Method	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre-processing	1.242	0.225	0.478	0.549	0.324	0.099	2
Inters. detec.	1.444	0.152	0.015	0.385	0.040	0.018	9
Total time	2.686	0.377	0.493	0.934	0.364	0.117	3
# Inters. tests	300,039	300,039	70,332	300,039	300,039	300,039	-
Datasets	UsCounty and UsAquifers						
Method	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre-processing	7.884	0.812	2.628	1.710	0.392	0.164	5
Inters. detec.	42.816	4.059	0.023	11.198	0.612	0.096	42
Total time	50.700	4.871	2.651	12.808	1.004	0.260	19
# Inters. tests	12,756,283	12,756,283	158,653	12,756,283	12,756,283	12,756,283	-
Datasets	UsCounty and UsCountyRotated						
Method	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre-processing	14.532	1.422	7.482	2.798	0.454	0.251	6
Inters. detec.	675.616	63.677	1.027	194.918	9.422	1.367	47
Total time	690.148	65.099	8.509	197.716	9.876	1.718	40
# Inters. tests	216,542,974	216,542,974	11,254,031	216,542,974	216,542,974	216,542,974	-

Table 2 – Times (in seconds) spent by the different version of the algorithms for 3 pairs of datasets. Column Speedup shows the speedup of the GPU method when compared against the sequential implementation (Interval*).

(thus, the number of pairs of segments that really need to be checked for intersection is smaller in the CGAL algorithm). However, this happens at a cost of a more expensive pre-processing step (up to 5 times slower than Interval*). Besides having a faster pre-processing step, the Interval* method can be parallelized, while CGAL is sequential.

Indeed, while the total processing-time of Interval* was from 1.3 times faster to 7.7 times slower than CGAL, the parallel version using the GPU had a speedup ranging from 4 times to 10 times.

To better understand the influence of the GPU on the results, consider the intersection of segments from UsCounty with UsCountyRotated as example. The total time spent by the Interval* implementation for detecting intersections is 63.677s (0.685s to prepare the predicates and 62.992s to evaluate them) and the time spent by the GPU implementation is 1.367s (1.149s to prepare the predicates and transfer the data to/from the GPU and 0.218s to perform the evaluation). If only the time to evaluate the intersection predicates is considered, the achieved speedup is 289×. This suggests that algorithms requiring a heavy usage of geometric predicates could benefit even more from the techniques presented in this chapter.

As expected, the number of failures of the intervals was equal on the CPU and on the GPU. In the intersection of BrSoil with BrCounty, only 4 of the 877 thousand evaluated predicates (0.0005%) failed, requiring an exact re-evaluation. In the intersection of UsCounty with UsAquifers, 3 of the 13 million predicates (0.00002%) failed. Finally, in the intersection of UsCounty with UsCountyRotated, 4 of the 224 million predicates failed (0.000002%).

2.5 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 line segments has been implemented. Because of the high computing power of the GPU for processing floating-point numbers, a speedup of up to 289 times (when compared against the sequential version) was obtained in the evaluation of the predicates (the speedup of the algorithm was up to 40 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 and computational geometry).

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 and MICs such as the Intel Xeon Phi 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).

3 Employing GPUs to accelerate exact geometric predicates for 3D geospatial processing

This chapter presents a technique to use GPUs to accelerate the computation of 3D geometric predicates. A common predicate is computing the orientation of four 3D points, which is a subproblem in applications such as intersecting two 3D meshes. Since the higher level application may require billions of evaluations, efficiency is important. Accuracy is required since floating roundoff errors can cause topological impossibilities. One solution is to compute with rational numbers, but that is difficult to implement on a GPU because rationals' sizes vary. Our solution is to compute on the GPU with interval arithmetic, but fall back to using rationals on the CPU if the interval computed on the GPU includes the origin; i.e., its sign is unknown. Our experiment with a dataset of hard rock mining drill holes show that this fallback to the CPU is rarely necessary; so that our technique gave a 17 times speedup compared to a sequential implementation.

3.1 Introduction

This chapter describes our experience using GPUs to accelerate the evaluation of exact geometric predicates, which are fundamental to many 2D and 3D applications in Geographic Information Science.

Floating point arithmetic presents a major challenge to computational geometry. Due to its finite representation, calculations are prone to round-off errors, which could lead to topologically incorrect results. For example, errors in a map overlay algorithm may generate polygons with slivers and self intersections.

A way to mitigate the problem is to represent geometric coordinates with arbitrary-precision rational numbers. This method, however, has several drawbacks when it comes to processing speed, since the computations can't rely on specialized hardware. Also, the number of digits needed to represent each value tend to grow as arithmetic operations are accumulated, which further degrades the performance.

However, arithmetic expressions in geometric predicates do not need to always be evaluated exactly in order to ensure the predicate will be computed correctly. Consider, for example, the 2D orientation predicate, which determines whether three points are collinear, make a left turn, or make a right turn. It can be computed by evaluating the sign of a determinant. Thus to exactly evaluate this predicate it is enough to guarantee

the sign of its determinant is computed correctly (its exact value is not important).

A technique often employed to perform this exact evaluation is the use of arithmetic filtering with interval arithmetic. The idea is to represent each value using intervals of floating-points numbers to bound the errors in the computation. If the exact result can be inferred from the intervals, it is returned. Otherwise, it is recomputed using higher precision.

In (MENEZES et al., 2019), we employed this technique for accelerating the detection of intersections between pairs of segments in 2D. This led to a speedup of 40 times compared to the sequential implementation. To accelerate the tests, a uniform grid is employed for indexing, and then an array containing the pairs to be checked for intersection is constructed and dispatched to the GPU, which performs the intersection tests with interval arithmetic.

A trivial extension of this technique to 3D did not yield a similar performance, and so in this chapter we will present a technique to circumvent this challenge.

3.2 Exact computation using Interval Arithmetic

Geometric algorithms are typically composed of predicates and constructions (operations that construct new values or coordinates). For example, a polygonal map overlay algorithm could employ predicates to perform point location and to detect pairwise intersections among the edges of the input polygons. Once pairs of intersecting edges are found, constructions could be employed to generate the coordinates of the new vertices created from the intersections.

Algorithms could be designed to not have any exactness guarantees (i.e., perform all the computation with floating-point numbers); to employ exact predicates (the results will be combinatorially correct, but the coordinates of the resulting geometric objects may have roundoff errors), but inexact constructions; and to have both exact predicates and exact constructions. The CGAL Geometric library (The CGAL Project, 2016), for example, allows the user to choose algorithms from these three categories.

The focus of this chapter is to describe an extension to our technique to accelerate exact geometric predicates. This could also be employed to accelerate algorithms with exact constructions (since these algorithms usually also rely on predicates).

One approach to ensure exactness consists in using only exact computation in the predicates. E.g., we might replace floating-point variables with multiple-precision rational numbers. The drawback is the performance overhead added by this technique (PION; FABRI, 2011). Furthermore, it is harder to parallelize exact algorithms because of the frequent memory allocations required as the rationals' precisions grow. Finally, imple-

menting these predicates on GPUs is also a challenge since there is apparently no mature library for dealing with multiple-precision rationals on GPUs.

An alternative technique is the use of arithmetic filtering with interval arithmetic. The idea is to represent each value v as an interval of floating point values that contains v. The arithmetic operations are then performed on the intervals. During each operation, the resulting interval is adjusted to guarantee it will contain the value that would be obtained if the expression was evaluated exactly (this is known as the *containment property*). In the computation of a predicate, if the sign of the exact result can be inferred from the sign of the resulting interval (i.e., if the lower bound of the interval is positive or the upper bound is negative), then this value is returned. Otherwise (this is known as filter failure), if the lower bound of the interval is negative and the upper bound is positive (which means the sign of the exact value could be negative, neutral or positive), the predicate is re-evaluated using exact arithmetic. That is slower but should rarely be necessary.

Key to ensuring the containment property is the set of guarantees provided by the IEEE-754 floating-point standard, which is widely adopted by processor manufacturers. This standard defines three rounding modes for floating-point operations. The result of each operation is guaranteed to be the result that would be obtained if it was computed exactly, and then rounded to the next (i.e., towards $+\infty$), previous (i.e., towards $-\infty$) or closest representable floating-point number. These first two rounding modes can be employed to ensure the containment property.

For example, consider the following predicate, which checks if the sum of two numbers is positive: $positiveSum(a, b) = (a + b > 0)$. Assuming the exact values of a and b are contained in the intervals $[a.lb, a.ub]$ and $[b.lb, b.ub]$, then the interval containing the exact value of $result = a + b$ could be created by performing the following operation: $result = [\lfloor a.lb{+}b.lb \rfloor, \lceil a.ub{+}b.ub \rceil]$, where $\lfloor\ \rfloor$ and $\lceil\ \rceil$ represent, respectively, the rounding to the previous and next representable floating-point values. Thus, a filtered version of $positiveSum(a, b)$ could be implemented as seen in Listing 3.1. Details about evaluating other operations using intervals are described in (BRÖNNIMANN; BURNIKEL; PION, 2001).

In this example, *computeExactPositiveSum* represents a function using exact arithmetic to evaluate the sign of $a.exact{+}b.exact$ (the $.exact$ is the exact representation of the values). For example, it could employ multiple precision rationals to do this computation.

Even though computations with intervals are faster than with rationals, they still add an overhead compared with simple floating-point arithmetic. This can be explained because computations with intervals require storing their two bounds and, also, performing more arithmetic operations, tests and changes in the floating-point rounding mode (as illustrated in the example for evaluating the sign of a sum in Listing 3.1). We propose employing GPUs to accelerate this. One advantage is that GPUs are particularly efficient

Listing 3.1 – Implementation of positiveSum

```
bool positiveSum(a,b) {
    result = [⌊a.lb + b.lb⌋, ⌈a.ub + b.ub⌉]
    if(result.ub < 0)
        // exact result guaranteed to be negative
        return false;
    else if(result.lb > 0)
        // exact result guaranteed to be positive
        return true;
    else
        //filter failure
        return computeExactPositiveSum(a.exact,b.exact);
}
```

for floating-point computation. NVIDIA GPUs have had functions to choose how floats are rounded since compute capability 1.3 for single precision floating point, and capability 2.0 for double precision (WHITEHEAD; FIT-FLOREA, 2011) around 2011.

3.3 Exact and efficient evaluation of predicates

In this section we present the main ideas behind this Gem. Subsection 3.3.1 gives an overview of the proposed technique. Subsection 3.3.2 exemplifies a GPU implementation of interval arithmetic, which is the basis of our work. We then proceed to explain how to deal with interval errors in Subsection 3.3.3. For a better understanding of our method, we describe in Subsections 3.3.4 and 3.3.5 two case studies efficiently employing our techniques. We close with a discussion on strategies for improving GPU performance, and some challenges faced while crafting this Gem.

3.3.1 The general strategy

Our technique is suited for geometric problems where many predicates need to be evaluated, and the inputs are known beforehand. Examples include computing boolean operations between maps, performing point location, and performing closest objects queries. The method consists of three main steps, which are explained next.

First, we must evaluate which input elements need to be checked. For example, when counting the number of intersections between the triangles of two meshes, we need to select which pairs of triangles to test. This is a combinatorial step, which can be accelerated by preprocessing the data with techniques such as spatial data structures to cull pairs of distant elements.

Once the data is ready, we can employ the parallel processing power of the GPU to accelerate the predicate computation. Interval arithmetic can be easily implemented

on the device (GPU) as will be shown in Section 3.3.2. Section 3.4 will give preliminary experiments to show that most predicates can be computed by interval arithmetic, and only a few pairs will need to be reevaluated with exact arithmetic.

The third and last step is to reevaluate the tests that failed. This part is usually done on the CPU side using arbitrary precision rational numbers, which guarantees an exact result. Even though rational numbers are costly, most pairs will be successfully evaluated on the GPU, so this step is not expected to affect performance in a significant way.

The strategy for sending the data from the CPU to the GPU is our main concern in this chapter. In our previous work (MENEZES et al., 2019), which is focused on intersecting 2D maps, an array of pairs to be tested is sent to the GPU, and each pair is tested by a thread. However, the process of creating the array is costly, and became the bottleneck of the algorithm when extended to 3D. Subsection 3.3.5 explains the methods used to circumvent the problem.

3.3.2 Implementing Arithmetic Filters on the GPU

As seen in Section 3.2, the floating point rounding modes need to be adjusted according to the desired computation. We adopt the same idea as employed by (COLLANGE; DAUMAS; DEFOUR, 2012) and (MENEZES et al., 2019), where a separate class is responsible for the interval arithmetic computations, making the predicate code clean and easy to maintain. The rounding mode switches can be achieved through compiler intrinsics, as illustrated in Listing 2.1.

Because of the GPU architecture and communications cost, when accelerating geometric algorithms it is important to group and process in batch the geometric predicates. Thus, the GPU is employed as a co-processor to evaluate big batches of geometric predicates.

3.3.3 Dealing with interval failures

When an interval failure occurs, we need to reevaluate the predicate using either a higher precision representation, or an exact representation such as arbitrary precision rational numbers, which are better suited for the CPU. Also, as seen in Sections 3.2 and 3.3.1, the number of interval failures is expected to be small, so the GPU parallel processing would not present a significant speedup when reevaluating the predicates.

For these reasons, we perform the exact computation of interval failures on the CPU side. When the GPU finishes processing the batch, it returns an array containing every pair where an interval failure occurred.

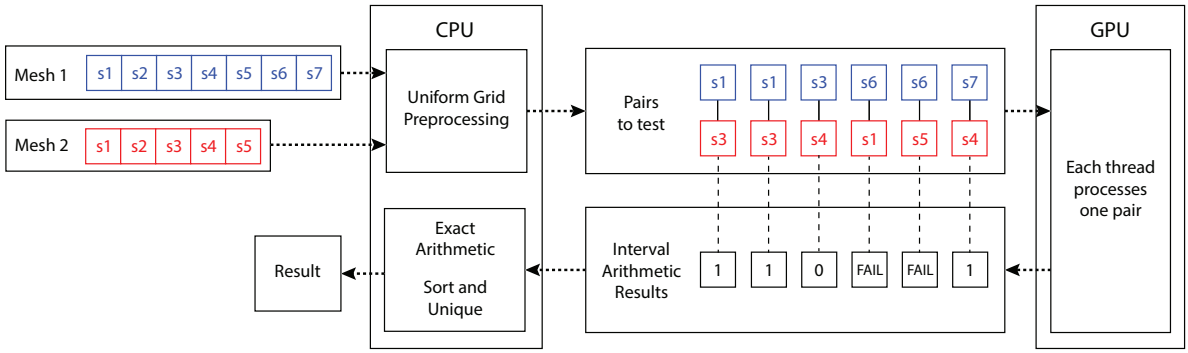


Figure 7 – Illustration of the algorithm employed for accelerating exact 2D segment intersections

3.3.4 Case study: 2D red/blue segment intersection

As a case study, in (MENEZES et al., 2019) we proposed a fast and exact algorithm for detecting intersections between two sets of 2D line segments (i.e., for reporting *red-blue* intersections). The algorithm consists of two steps: on the CPU a uniform grid is employed for indexing the sets. Each segment is inserted into the grid cells intersecting its bounding-box.

Then, an array of potentially intersecting pairs of edges (pairs of segments in the same cell) is generated and copied to the GPU, where they are tested for intersection. Each GPU thread is designed to evaluate one pair for intersection.

Each intersection is evaluated by computing 4 2D orientation predicates (MENEZES et al., 2019). The results are stored in an array and a flag is employed to indicate if an intersection evaluation could not be determined reliably due to an interval failure. Once the results are copied back to the CPU, the (hopefully few) unreliable results are exactly re-evaluated using rationals. Considering the experiments performed in (MENEZES et al., 2019), in the worst case only 0.0005% of the predicates failed and required re-evaluation.

Since an edge may be inserted into multiple uniform grid cells, some intersecting pairs may be reported more than once. These duplicates are removed at the end of the algorithm since experiments showed that this was faster than ensuring the list of potentially intersecting pairs had only unique values (because this list is typically much larger than the list of actually intersecting pairs). Figure 7 illustrates the steps described above.

This method works well when the batch array is small enough that its creation on the CPU side does not impact the algorithm's overall performance. However, when this approach is extended to three dimensions, the number of tests grow fast, and the time needed for traversing the grid and creating the batch array quickly becomes the algorithm's bottleneck.

In this chapter we present a more efficient and robust approach for distributing work between GPU threads, since thread balancing is key for achieving efficiency on SIMT

architectures.

3.3.5 Case study: 3D segment-triangle intersection

In this section we will deal with the following problem as another case study: given a set of line segments and a set of triangles, detecting pairs of segments and triangles that do intersect (in this chapter we will often refer to them simply as *pairs*). One application is detecting intersections between triangles in a mesh intersection method. Another is detecting intersections between sampling drill holes (segments) and 3D objects representing mineral deposits in a mine (REAL et al., 2019).

The algorithm employed to check if a segment intersects a triangle is based on the evaluation of five 3D orientation predicates. Consider the example in Figure 8. The interior of edge DE will intersect the interior of triangle ABC if, and only if, D and E are on opposing sides of the plane defined by ABC (i.e., if the orientations of points (A, B, C, D) and (A, B, C, E) have opposing signs), and the orientations of (A, B, D, E), (B, C, D, E), and (C, A, D, E) have the same sign.

The 3D orientation of points (A, B, C, D) is given by the sign of the following determinant:

$$\begin{vmatrix} A_x & A_y & A_z & 1 \\ B_x & B_y & B_z & 1 \\ C_x & C_y & C_z & 1 \\ D_x & D_y & D_z & 1 \end{vmatrix}$$

As the intersection test consists in evaluating the signs of five determinants, we can employ arithmetic filters for a fast and exact computation. Also, before evaluating each pair, the algorithm performs a fast bounding-box test in order to further cull the number of tests being performed. In this section we present three different approaches for doing this computation on the GPU. Experiments comparing them will be presented at the end of the section.

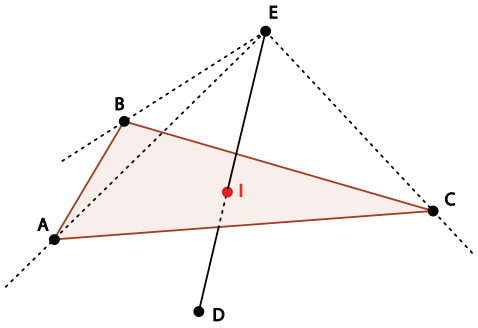


Figure 8 – Line segment—triangle intersection test; decomposed into five 3D orientation predicates.

We initially implemented a version of this algorithm using a strategy similar to the one described in Section 3.3.4 for 2D intersection. I.e., the list of potentially intersecting pairs of segments and triangles is extracted from the uniform grid on the CPU, and then copied to the GPU, where each one is evaluated by one thread. However, preliminary experiments showed that this pre-processing on the CPU was a bottleneck, and so other approaches were tested.

The second approach consists of creating the uniform grid and transferring to the GPU the list (using a ragged array) of triangles in each grid cell. Then, each GPU thread will process one segment s (or triangle t). The processing of s consists in iterating over all triangles (or segments) that are in the uniform grid cells intersecting the bounding-box of s and testing them for intersections with s (or triangle t).

The drawback of this second approach is that geometric data may be non-uniformly distributed, which makes the number of triangles potentially intersecting each segment vary. This leads to load unbalancing and thread divergence.

Finally, we employed a strategy that avoids the pre-processing step for generating the list of pairs of segments and triangles on the CPU and balances the amount of work done by each GPU thread. The basic idea is to implicitly generate on the GPU the pairs to be processed and assign one thread to process each one.

Let T_c and S_c be, respectively, the number of triangles and segments in the uniform grid cell c. Assume the kernel employed to compute the intersections is launched using Th threads per block. We launch several blocks for each uniform grid cell in order to ensure all pairs will be processed. Threads in the same block always process the same grid cell) and, thus, $\left\lceil \frac{T_c \times S_c}{Th} \right\rceil$ blocks are launched for each cell c.

In order for each thread to know which pair of segment and triangle it will process, 3 arrays are created (with size equal to the number of blocks) on the CPU and copied to the GPU:

- $Cell$: the cell being processed by the corresponding block.
- $First_{pair}$ and $Last_{pair}$: the index (within the cell) of the, respectively, first and last pairs of segment-triangle being processed by the block.

Thus, the thread with index tid in block b will process the pair $P = tid + First_{pair}[b]$ of cell $C = Cell[b]$. The index of the segment and triangle can be obtained by, respectively, the expressions $P\%S_c[C]$ and $P/S_c[C]$ (in this chapter we assume all indices are 0-based).

Tables 3 and 4 illustrates the distribution of pairs to the threads, supposing the uniform grid has 4 cells and each GPU block launches 4 threads. Notice that cell 0 has 3 triangles and 4 segments, therefore, it will require 3 blocks of threads to process its

$3 \times 4 = 12$ pairs. Cell 1, on the other hand, will require 2 blocks and cells 2 and 3 will require 1 block each. Thus, a total of 7 blocks will be created.

Table 4 describes the blocks (e.g.: blocks from 0 to 2 process cell 0). Consider, for example, the thread with id 2 of block 4. This thread will be responsible for processing pair $Start_{pair}[4]+1 = 5$ in cell $Cell[4] = 1$. Since this cell has 3 segments, this thread will evaluate the intersection between segment $5\%3 = 2$ and triangle $5/3 = 1$.

Table 3 – Uniform grid cells

Cell	0	1	2	3
T_c	3	2	1	3
S_c	4	3	1	1
Blocks	3	2	1	1

Table 4 – GPU blocks of threads

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

By employing this strategy, the blocks will evaluate the same number of intersections, except for the last block processing each cell (that may process fewer pairs if the number of pairs in that cell is not a multiple of the number of threads per block).

Another challenge is the strategy employed to report which pairs do intersect. The implementation that creates a list of pairs on the CPU employs a flag for each pair indicating whether it does intersect.

Assuming the number of intersections is much smaller than the number of potentially intersecting pairs, transferring such a list from the GPU in the other two implementations could become a bottleneck of the algorithm. Thus we employed a sparse list in that implementation (i.e., instead of using boolean flags we store in a list the pairs that do intersect, and use another list to report pairs with interval arithmetic failures).

A two-step strategy was employed to create the resulting array. First, a kernel is launched to count the number of intersections. Then the array is allocated and the kernel is launched again to actually insert the ids of the intersecting segments/triangles in it. An atomic fetch and increment operation is performed to avoid race conditions when inserting the data into the array.

For performance, since in the first step we only need to compute an upper bound for the number of resulting intersections, we employed a floating-point version of the algorithm and defined the size of the resulting array as 5% larger than the reported value. In the improbable case where the number of intersections is larger than this estimate, the algorithm detects this error and the resulting array is reallocated with a larger capacity (in this situation the kernel using interval arithmetic needs to be launched again). In all our experiments no case of underestimated array size has been experienced.

3.4 Experiments

We performed experiments on an AMD Ryzen 5 1600 machine with 6 cores at 3.2 GHz (12 hyperthreads), 16GB of RAM, and a NVIDIA Geforce GTX 1070Ti GPU (8 GB of RAM). These experiments were performed using the mining dataset provided by (REAL et al., 2019), where triangles were extracted from the geological shapes representing minerals and segments represent the drill holes in the mine.

This dataset has 1 000 000 triangles and 7846 segments. We used a 100^3 uniform grid to index the data, which generated 15 453 302 pairs of potentially intersecting segments and triangles. The final number of intersections reported by the algorithms was 211 thousand.

Table 5 presents the times spent by the following five strategies: a sequential CPU implementation (which iterates through the uniform grid cells and evaluates each pair for intersection), an implementation that creates the list of potentially intersecting pairs on the CPU and processes them on the GPU, two implementations that launches one thread per segment (resp. triangle) and test them for intersection against triangles (resp. segments) in uniform grid cells intersecting its bounding-box. Finally, the last algorithm is the implementation proposed in this work, that employs multiple thread blocks to process each cell.

As mentioned before, creating a list of pairs equalizes the work done by each thread (improving the load balancing), but has high pre-processing and memory overheads (only 6% of the time is actually spent testing the intersections).

On the other hand, assigning one segment or triangle per thread reduces the pre-processing cost (since a list of pairs is not generated), but the intersection step is slower due to the load unbalance. It is interesting to notice that the intersection step is 9 times faster when each thread is responsible for a triangle than when it is responsible for a segment. This is because this sample dataset has 127 times more triangles than segments and, thus, the first version uses more threads (and each one does less work), which is usually preferred on GPUs and improves load balancing.

Finally, the best result is obtained using multiple blocks per grid cell. It combines

Table 5 – Time (in seconds) spent by 5 different strategies for intersecting triangles and segments.

	Sequential	List of pairs	One segment/thread	One triangle/thread	Mult. blocks/cell
Pre-processing	-	0.67	0.00	0.00	0.00
Mem. (CPU to GPU)	-	0.21	0.02	0.01	0.02
Intersection	2.48	0.09	5.96	0.69	0.10
Mem. (GPU to CPU)	-	0.02	0.00	0.00	0.00
Rationals	0.00	0.03	0.00	0.00	0.00
Ensure uniqueness	0.01	0.01	0.02	0.02	0.02
Total	2.50	2.14	6.00	0.72	0.15

a fast pre-processing with a small intersection detection time on the GPU. Indeed, its intersection step performs two passes (as explained in Section 3.3.5), spending 0.01 second in the first pass (done with floating-point arithmetic) and 0.09 second in the second (performed with interval arithmetic). This result suggests that this implementation has good load balancing, since the version using a list of pairs (where all threads perform the same amount of work) also spent 0.09 second detecting intersections.

In all preliminary experiments the time spent using rationals to process the cases with filter failure was negligible (in the experiment with the most failures, only 0.0005% of the predicates required re-evaluation with rationals). In this experiment, only 19 of the 15 453 302 intersection tests had a filter failure.

The fastest proposed implementation had a speedup of 17 times compared to the sequential implementation. If only the intersection time is considered, then the speedup is 25×.

The bounding-box filter employed before each intersection evaluation significantly reduces the intersection time on the CPU (if the bounding-box was not employed, this time would be 7.73s instead of 2.48s). This difference is smaller on the fastest GPU version (both the version with and the one without the bounding-box tests took 0.10s). Indeed, while the bounding-box reduces the amount of computation, it creates thread divergence when some threads in the same warp pass the test while others fail. If this test was not employed (in both the CPU and GPU versions), the GPU speedup would have been 77×. This suggests algorithms where this filtering could not be applied (or with more expensive computations after the bounding-box tests) could benefit even more from the techniques presented in this chapter.

3.5 Conclusion and Future Work

This chapter presented a fast approach for accelerating exact geometric algorithms using GPUs. While GPUs cannot be easily employed to perform exact computation with multiple-precision rational numbers, they can be employed to accelerate exact algorithms that employ arithmetic filters with interval arithmetic to avoid using (slow) rationals. In the experiments, we observed a speedup of 25× in the evaluation of the intersections with intervals (and 17 times if the total running-time of the algorithm is considered).

This performance allows the fast processing of big geographical datasets (such as 2D maps or mining models), while ensuring geometrical degeneracies caused by roundoff errors are avoided.

As future work, we intend to further improve our implementation, by utilizing faster memories (such as the shared memory) better, and by using techniques such as streaming to overlap computation and memory operations. Furthermore, some ideas ap-

plied in our 3D implementation have not been implemented in the 2D version of the problem, and thus implementing them is also an interesting direction of future work.

4 Fast parallel evaluation of exact geometric predicates on GPUs

This chapter presents new extensions and improvements on the previous techniques 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.

4.1 Introduction

In this chapter, 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 (MENEZES et al., 2019) (this version is presented in chapter 2 of this dissertation). 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 (MENEZES et al., 2020), we extended the previous algorithm for computing intersections between segments and triangles in 3D ((chapter 3 of this dissertation)). 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 (MENEZES et al., 2020) 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 chapter, 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 chapter 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 4.2. 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.

4.2 Fast red-blue intersection tests

To evaluate the improvements proposed in this chapter, 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). 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.

4.2.1 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 (MAGALHÃES; FRANKLIN, 2017). 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 (AKMAN et al., 1989; FRANKLIN et al., 1988; HOPKINS; HEALEY, 1990).

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. (MAGALHÃES; FRANKLIN; ANDRADE, 2017), 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 (MAGALHÃES; FRANKLIN, 2017). The entire structure being in contiguous memory also makes it easier to transfer the grid to the GPU. Figure 5 illustrates the indexing of a mesh using a 2D uniform grid built with a dynamic array (a), and a ragged array (b).

4.2.2 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 (MENEZES et al., 2020). In order to avoid this costly pre-processing step, an approach for implicitly associating a thread with a pair will be employed.

4.2.3 Implicit Thread Association

In (MENEZES et al., 2020) 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 chapter 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 (MENEZES et al., 2020)).

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, $\left\lceil \frac{T_{1c} \times T_{2c}}{T_c} \right\rceil$ 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 (MENEZES et al., 2020)) illustrate how threads are associated to the triangles they are processed:

Table 6 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 7 presents the arrays $Cell$, $First_{pair}$ and $Last_{pair}$ created for the uniform grid from Table 6. 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 are configured to process the same number of pairs of triangles (except for the last block processing a cell, which may evaluate fewer pairs).

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

Table 6 – Uniform grid cells. Adapted from: (MENEZES et al., 2020)

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 7 – GPU blocks of threads. Adapted from: (MENEZES et al., 2020)

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 (MENEZES et al., 2020), 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.

4.2.4 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 (SEGURA; FEITO, 2001).

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.

4.2.5 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 (HOBEROCK; BELL, 2010).

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 (SHEWCHUK, 1997).

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

4.3 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 (GRANLUND; the GMP development team, 2014).

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 (MAGALHÃES; FRANKLIN, 2017; AUDET et al., 2013). 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. (MAGALHÃES; FRANKLIN, 2017), the range of grid configurations with reasonable

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 8 – Number of vertices and triangles in the meshes employed in the experiments.

performance optimum is broad.

Experiments have been performed using meshes available in two public repositories. The Armadillo mesh was downloaded from the Stanford repository (The Stanford 3D Scanning Repository, 2016) and the other ones were downloaded from Thingi10K (ZHOU; JACOBSON, 2016). These meshes have been tetrahedralized using GMSH (GEUZAINE; REMACLE, 2009) before being employed in the experiments. Table 8 describes the dataset and Figure 9 illustrates it. As it can be seen, the size of these datasets ranged from 600 thousand to 8 million triangles. Figure 10 shows the interior of mesh 914686 after tethraderalization, and Figure 11 shows two overlaid pairs of meshes employed in the experiments.

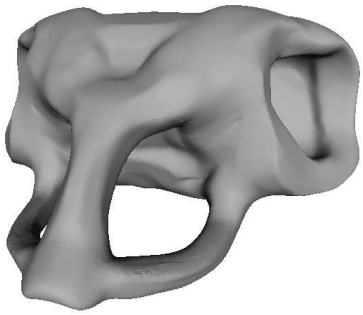
Differently from CPUs, GPUs typically have significantly more single-precision floating point units than double-precision (since they are mainly focused on applications where the lower precision of floats is acceptable) (SUN et al., 2019). For example, the RTX 8000 GPU employed in these experiments have 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 4.2.5, 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.

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

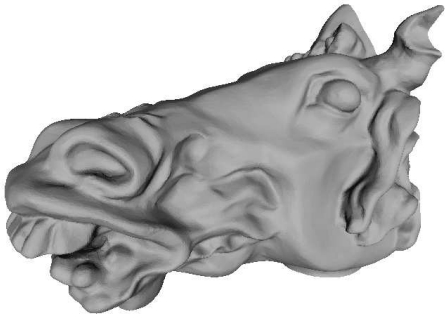
- *CPU*: sequential CPU implementation employing arithmetic filters using double-precision intervals (provided by the CGAL Interval_nt class (The CGAL Project, 2016)). This is the baseline implementation.
- *GPUDouble*: parallel GPU implementation using double-precision intervals implemented as described in this chapter.
- *GPUFloat*: the same implementation as GPU double, but employs single-precision floats in the intervals.

(a) Armadillo

(b) 914686

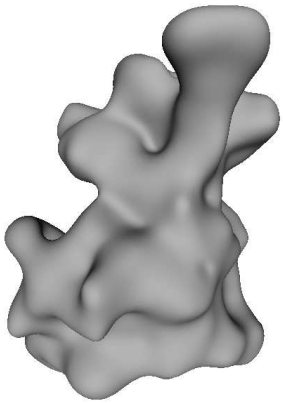


(c) 260537



(d) 68380

(e) 518092



(f) 461112

Figure 9 – Meshes employed in the experiments.

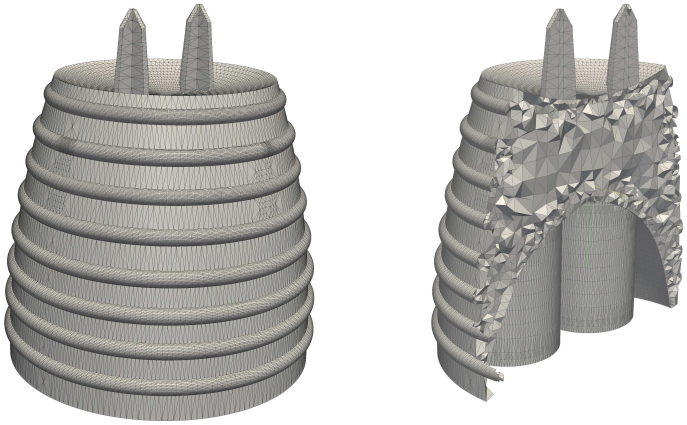
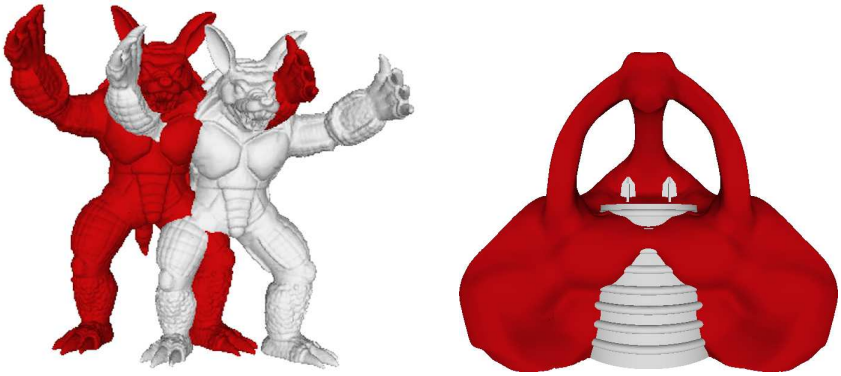


Figure 10 – Mesh 914686 (left). The right figure illustrates its tetrahedralized interior (Source: (MAGALHÃES; FRANKLIN, 2017)).



(a) (b)

Figure 11 – Examples of pairs selected for intersection tests. (a) Armadillo (red) and ArmadilloTranslated (white) (b) 260537 (red) and 914686 (white).

4.3.1 Experiments on the RTX 8000 GPU

Tables 9 and 10 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 4.2.3).

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, inter-

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 9 – 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.

mediate 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

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 x 10^6			
#intersection tests		72.8 x 10^6			
#intersections		16.8 x 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 x 10^6			
#intersection tests		505.4 x 10^6			
#intersections		17.8 x 10^6			

Table 10 – Times (in seconds) spent by the the intersection tests between meshes Armadillo vs ArmadilloTranslated and 461112 vs 518092. The Speedup columns shows the speedup of the corresponding GPU version of the algorithm when compared against the CPU one.

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.

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 11 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 9 and 10, 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).

4.3.2 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 lower-end GPU. Table 12 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).

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 11 – 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.

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 12 – 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.

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*.

4.4 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).

5 General Conclusion

In this work we proposed a fast, exact and parallel method for efficient geometric predicate evaluation using the GPU. The core of our idea is to compute the predicates on the GPU using interval arithmetic, and the (few) results that cannot be exactly calculated are reevaluated on the CPU with exact arithmetic.

The framework was built incrementally, and at each step a solution was proposed for the current bottleneck. To measure efficiency, we implemented algorithms for computing intersections between geometric objects. Additional filtering passes were employed, such as culling the input with a uniform grid, and a bounding-box filtering step to further cull pairs of distant objects.

The latest version of our algorithm, when tested for detecting intersections between pairs of triangles in 3D, achieved speedups of up to 1936× for intersection evaluation, and up to 414× considering the overall time (when compared against a sequential implementation on the CPU). The technique proved to be efficient, while still maintaining exactness, which makes our framework viable for exact geometric computations in interactive applications (fields such as CAD, GIS, and 3D modeling could benefit from our method).

The current bottleneck of our algorithm is in the memory transfers between CPU and GPU. This suggests that algorithms that perform a higher number of computations per data could benefit even more from our framework. This also leaves room for future improvements, and techniques such as using pinned memory to improve the transfer rates could further increase the speedups.

As this was an incremental work, the enhancements developed in its course were not tested in the previous case studies (2D segment intersection and 3D segment-triangle intersection). We expect the modifications performed on the last step to significantly improve the first ones.

Another interesting future experiment is to test how the method would scale when using multiple GPUs. The algorithm could be easily adapted for this case, and we believe it should present even better speedups, especially when using huge datasets, which would require more memory on the graphics card. It would also allow the algorithm to perform computations on multiple datasets at the same time.

Finally, we intend to perform more thoroughly experiments analysing the GPU resource usage, using tools such as *nvprof*. This should give insightful directions on where to proceed in terms of GPU performance, which would be invaluable, should the memory transfer bottleneck be reduced with future optimizations.

Bibliography

AKMAN, V. et al. Geometric computing and the uniform grid data technique. **Comput. Aided Des.**, v. 21, n. 7, p. 410–420, Sept. 1989.

AUDET, S. et al. Robust and efficient polygon overlay on parallel stream processors. In: **Proc. 21st ACM SIGSPATIAL Int. Conf. Advances Geographic Information Systems**. New York, NY, USA: ACM, 2013. (SIGSPATIAL'13), p. 304–313. ISBN 978-1-4503-2521-9.

BELUSSI, A. et al. Snap rounding with restore: An algorithm for producing robust geometric datasets. **ACM Trans. Spatial Algorithms and Syst.**, ACM, New York, NY, USA, v. 2, n. 1, p. 1:1–1:36, mar. 2016. ISSN 2374-0353.

BERG, M. de; HALPERIN, D.; OVERMARS, M. An intersection-sensitive algorithm for snap rounding. **Computational Geometry**, Elsevier, Amsterdam, The Netherlands, v. 36, n. 3, p. 159–165, Apr. 2007.

BRÖNNIMANN, H.; BURNIKEL, C.; PION, S. Interval arithmetic yields efficient dynamic filters for computational geometry. **Discrete Applied Mathematics**, Elsevier, v. 109, n. 1-2, p. 25–47, 2001.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. **Professional CUDA C Programming**. 1st. ed. GBR: Wrox Press Ltd., 2014. ISBN 1118739329.

COLLANGE, S.; DAUMAS, M.; DEFOUR, D. Chapter 9 - interval arithmetic in CUDA. In: HWU, W. mei W. (Ed.). **GPU Computing Gems Jade Edition**. Boston: Morgan Kaufmann, 2012, (Applications of GPU Computing Series). p. 99 – 107. ISBN 978-0-12-385963-1.

COLLANGE, S.; FLÓREZ, J.; DEFOUR, D. A gpu interval library based on boost.interval. In: **8th Conference on Real Numbers and Computers**. [S.l.: s.n.], 2008. p. 61–71.

European Space Agency. **Ariane 501 inquiry board report**. Paris: [s.n.], 2015. (Retrieved on 06/15/2015). Disponível em: <ravel.esrin.esa.it/docs/esa-x-1819eng.pdf>.

FRANKLIN, W. R. et al. Efficiency of uniform grids for intersection detection on serial and parallel machines. In: MAGNENAT-THALMANN, N.; THALMANN, D. (Ed.). **New Trends in Computer Graphics (Proc. Computer Graphics Int.'88)**. Berlin, Germany: Springer-Verlag, 1988. p. 288–297.

GEUZAINE, C.; REMACLE, J.-F. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. **Int. J. for Numerical Methods in Eng.**, John Wiley Sons, Ltd., v. 79, n. 11, p. 1309–1331, maio 2009.

GRANLUND, T.; the GMP development team. **GNU MP: The GNU Multiple Precision Arithmetic Library**. 6.0.0. ed. [S.l.], 2014. <http://gmplib.org/> (Retrieved on 10/19/2017).

HERSHBERGER, J. Stable snap rounding. **Comput. Geom.**, Elsevier, Amsterdam, The Netherlands, v. 46, n. 4, p. 403–416, May 2013.

HOBBY, J. D. Practical segment intersection with finite precision output. **Comput. Geom.**, Elsevier, Amsterdam, The Netherlands, v. 13, n. 4, p. 199–214, Oct. 1999.

HOBEROCK, J.; BELL, N. **Thrust: A Parallel Template Library**. [S.l.], 2010. "Version 1.7.0", <http://thrust.github.io/> (Retrieved on 10/19/2017).

HOFFMAN, C. M. The problems of accuracy and robustness in geometric computation. **Comput.**, IEEE Computer Society, Los Alamitos, CA, USA, v. 22, n. 3, p. 31–40, Mar. 1989. ISSN 0018-9162.

HOPKINS, S.; HEALEY, R. G. A parallel implementation of Franklin's uniform grid technique for line intersection detection on a large transputer array. In: BRASSEL, K.; KISHIMOTO, H. (Ed.). **4th Int. Symp. Spatial Data Handling**. Zürich: [s.n.], 1990. p. 95–104.

JACOBSON, A.; PANOZZO, D. et al. **libigl: A Simple C++ Geometry Processing Library**. [S.l.], 2016. <http://libigl.github.io/libigl/> (Retrieved on 10/18/2017).

JOLDES, M. et al. Campary: CUDA multiple precision arithmetic library and applications. In: SPRINGER. **International Congress on Mathematical Software**. [S.l.], 2016. p. 232–240.

KETTNER, L. et al. Classroom examples of robustness problems in geometric computations. **Comput. Geom.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, v. 40, n. 1, p. 61–78, May 2008. ISSN 0925-7721.

LI, C.; PION, S.; YAP, C. K. Recent progress in exact geometric computation. **The J. Log. Algebr. Program.**, Elsevier, Amsterdam, The Netherlands, v. 64, n. 1, p. 85–111, July 2005.

LU, M.; HE, B.; LUO, Q. Supporting extended precision on graphics processors. In: ACM. **Proceedings of the sixth international workshop on data management on new hardware**. [S.l.], 2010. p. 19–26.

MAGALHÃES, S. V.; FRANKLIN, W. R.; ANDRADE, M. V. Fast exact parallel 3d mesh intersection algorithm using only orientation predicates. In: ACM. **Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems**. [S.l.], 2017. p. 44.

MAGALHÃES, S. V. G.; FRANKLIN, W. R. **Exact and parallel intersection of 3d triangular meshes**. Tese (Doutorado) — Rensselaer Polytechnic Institute, USA, 2017.

MENEZES, M. et al. Employing gpus to accelerate exact geometric predicates for 3d geospatial processing. In: KRUMM, J. (Ed.). **2nd ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2020)**. ACM, 2020. Disponível em: <https://www.spatialgems.net/>.

MENEZES, M. de M. et al. Accelerating the exact evaluation of geometric predicates with GPUs. In: SHONTZ, S.; PEIRO, J.; VIERTEL, R. (Ed.). **28th International Meshing Roundtable**. Buffalo, NY, USA: [s.n.], 2019. ISBN 978-1-7334890-0-3.

MULLER, J.-M. et al. **Handbook of Floating-Point Arithmetic, 2nd edition**. [S.l.]: Birkhäuser Boston, 2018. 632 p. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9.

NVIDIA; VINGELMANN, P.; FITZEK, F. H. **CUDA Toolkit Documentation v11.2.1**. 11.2.1. ed. [S.l.], 2021. <https://docs.nvidia.com/cuda/> (Retrieved on 08/03/2021).

OLIVEIRA, M. A. et al. Evaluating the usage of exact queries on 3d spatial databases. In: **Proc. XXI Brazilian Symp. Geoinformatics**. [S.l.: s.n.], 2020. (GeoInfo '20), p. 162 – 167.

PION, S.; FABRI, A. A generic lazy evaluation scheme for exact geometric computations. **Sci. Comput. Program.**, v. 76, n. 4, p. 307 – 323, Apr. 2011. ISSN 0167-6423.

POPESCU, V. **Towards fast and certified multiple-precision librairies**. Tese (Doutorado) — Université de Lyon, 2017.

REAL, L. C. V. et al. Large-scale 3d geospatial processing made possible. In: **Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGSPATIAL '19), p. 199–208. ISBN 9781450369091. Disponível em: <https://doi.org/10.1145/3347146.3359351>.

SEGURA, R. J.; FEITO, F. R. Algorithms to test ray-triangle intersection. comparative study. In: **The 9-th Int. Conf. Central Europe Comput. Graph., Visualization Comput. Vision'2001, WSCG 2001**. [S.l.: s.n.], 2001. p. 76–81.

SHEWCHUK, J. R. Adaptive precision floatingpoint arithmetic and fast robust geometric predicates. **Discret. & Comput. Geom.**, v. 18, n. 3, p. 305–363, Oct. 1997.

SKEEL, R. Roundoff error and the patriot missile. **SIAM News**, v. 25, n. 4, p. 11, July 1992.

SUN, Y. et al. Summarizing cpu and gpu design trends with product data. **arXiv preprint arXiv:1911.11313**, 2019.

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

The Stanford 3D Scanning Repository. **"The stanford 3D scanning repository"**. 2016. (Retrieved on 10/19/2017). Disponível em: <http://graphics.stanford.edu/data/3Dscanrep/>.

WHITEHEAD, N.; FIT-FLOREA, A. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. **rn (A+ B)**, v. 21, n. 1, p. 18749–19424, 2011.

YAP, C. K. Towards exact geometric computation. **Comput. Geom.**, Elsevier, Amsterdam, The Netherlands, v. 7, n. 1–2, p. 3 – 23, Jan. 1997. ISSN 0925-7721.

ZHOU, Q.; JACOBSON, A. Thingi10k: A dataset of 10,000 3d-printing models. **arXiv preprint arXiv:1605.04797**, 2016.