ACCELERATING THE EXACT EVALUATION OF GEOMETRIC PREDICATES WITH GPUS

Marcelo de Matos Menezes[1] Salles Viana Gomes de Magalhães[2]
Matheus Aguilar de Oliveira[3] W. Randolph Franklin[4]
Rodrigo Eduardo de Oliveira Bauer Chichorro[5]

[1]*Universidade Federal de Viçosa (MG) Brasil, marcelo.menezes@ufv.br*
[2]*Universidade Federal de Viçosa (MG) Brasil, salles@ufv.br*
[3]*Universidade Federal de Viçosa (MG) Brasil, matheus.a.aguilar@ufv.br*
[4]*Rensselaer Polytechnic Institute, Troy NY, USA, mail@wrfranklin.org*
[5]*Universidade Federal de Viçosa (MG) Brasil, rodrigo.chichorro@ufv.br*

ABSTRACT

This paper presents a technique for employing high-performance computing for accelerating the exact evaluation of geometric predicates. Arithmetic filters are implemented using interval arithmetic to reduce the necessity of exact arithmetic while ensuring the results of the predicates are still exact. Furthermore, the computation with interval arithmetic is offloaded to a CUDA-enabled GPU. If the GPU detects that some results cannot be trusted, the corresponding predicates are re-evaluated in parallel on the CPU using arbitrary-precision rational numbers. As a case study, a red-blue 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.

Keywords: computational geometry, exact computation, high-performance computing, GPU, CUDA

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 [1], most of the time computation with intervals is enough to infer the exact result and, thus, predicates can be efficiently and exactly evaluated without the overhead of exact computation.

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

In this paper, we propose the use of a combination of GPUs and multi-core CPUs to accelerate the evaluation of exact predicates using arithmetic filters. 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. BACKGROUND

2.1 Roundoff errors

Non-integer numbers are typically approximately represented in computers with floating-point values. The difference between the value of a non-integer number and its approximation is often referred as roundoff error. Even though these differences are usually small, these errors accumulate as sequences of arithmetic operations are performed. The presence of floating point errors in computer programs often creates serious consequences in diverse fields such as the failure of the first Ariane V rocket [2] and the failure of the Patriot missile defense system [3].

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. [4] 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. [4] implemented a program to apply the planar orientation predicate ($orientation(p, q, r)$) on a point $p = (p_x + xu, p_y + yu)$ where u is the step between adjacent floating point numbers in the range of p and $0 \leq x, y \leq 255$. This results in a 256×256 matrix containing either blue, yellow and red points meaning that the corresponding point is detected to be above, on or below the line that passes through q and r. Figure 1 shows the geometry of this experiment for $p = (0.5, 0.5)$, $u = 2^{-53}$, $q = (12, 12)$ and $r = (24, 24)$. As it can be seen, several points have their orientation computed incorrectly.

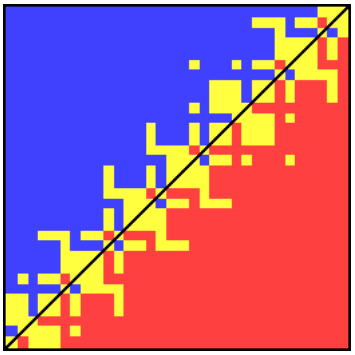


Figure 1: Roundoff errors in the planar orientation problem - Geometry of the planar orientation predicate for double precision floating point arithmetic. Yellow, red and blue points represent, respectively, collinear, negative and positive orientations. The diagonal line is an approximation of the segment (q, r). Source: [4].

As shown by [4], these inconsistent results in the orientation predicates could make algorithms that use this predicate to fail.

Some techniques have been proposed to handle this problem. The simplest one, the epsilon-tweaking, consists of using an ϵ tolerance that considers two values x and y are equal if $|x - y| \leq \epsilon$. However this is a formal mess because equality is no longer transitive, nor invariant under scaling. Thus, in practice, epsilon-tweaking fails in several situations [4].

Snap rounding is another method to approximate arbitrary precision segments into fixed-precision numbers [5]. 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 [6, 7, 8].

Shewchuk [9] 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 [9], 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" [9].

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 [10, 11, 4, 12]. Computing in the algebraic field of the rational numbers over the integers, with the integers allowed to grow as long as necessary, allows the traditional arithmetic operations, $+, -, \times, \div$, to be computed exactly, with no roundoff error.

The cost is that the number of digits in the result of an operation is about equal to the sum of the numbers of digits in the two inputs. E.g., $\frac{214}{433} + \frac{659}{781} = \frac{452481}{338173}$. Casting out common factors helps, but that is rarely possible. However, this behavior is acceptable if the depth of the computation tree is small. Also, the performance penalty associated with rationals can be significantly reduced by employing techniques such as arithmetic filtering with interval arithmetic, as we will discuss in section 2.2.

2.2 Arithmetic filters and interval arithmetic

One technique to accelerate algorithms based on exact arithmetic is to employ arithmetic filters and interval arithmetic [13]. 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" [13]. 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. [13] 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 [13].

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

The excellent Computational Geometry Algorithms Library (CGAL) [14] 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 [13]. Listing 1 illustrates one of the ways to develop an arithmetic filter using C++ and CGAL: variables with the suffix *_exact* were created as GMP[15] (*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).

Listing 1: Using CGAL interval arithmetic framework

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

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" [13]).

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 [16].

2.3 High-performance computing and CUDA

The advent of powerful multi-core CPUs and General Purpose GPUs (*GPGPUs*) with thousand of cores has increased the computing capability of relatively inexpensive computers. For example, currently (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 [17], performs (non-exact) map overlays using GPUs.

Zhou et al. [16] and Magalhães et al. [18] have developed parallel (for shared-memory multi-core CPUs) and exact algorithms for performing boolean operations on 3D meshes. Zhou et al. [16] 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. [18], 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" [19]. One of the reasons for this is the typically non-trivial memory management required by this kind of computation [20].

Thus, libraries for performing higher-precision arithmetic on GPUs (such as CAMPARI [20] and GARPREC [21]) are typically designed to process extended-precision floating-point numbers.

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

While there has been research [22, 23] on the field of implementing interval arithmetic on GPUs, these works have focused on computer graphics applications (like ray tracing) and have not employed this technique to accelerate exact geometric computation using arithmetic filters.

3. IMPLEMENTING EXACT PARALLEL PREDICATES

As stated in section 2.2, a correct implementation of interval arithmetic relies on hardware compliance

to the IEEE-754 standard. NVIDIA's GPUs double and single precision floating point implementations are in accordance with the standard since compute capabilities 1.3 and 2.0, respectively [24]. 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 [22].

In order to make interval arithmetic transparent during the evaluation of geometric predicates, we created a separate class, based on Collange et al. [22], 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. [22], the addition of two intervals $[a, b]$ and $[c, d]$ can be performed using the expression $[a, b] + [c, d] = [\underline{a + c}, \overline{b + d}]$ (where $\underline{a + c}$ and $\overline{b + d}$ indicate, respectively, the expression is rounded towards $-\infty$ and $+\infty$). Listing 2 illustrates the implementation of the addition method, where the CUDA C functions *__dadd_rd* and *__dadd_ru* switches the double precision floating point rounding mode for additions to $-\infty$ and $+\infty$, respectively.

Listing 2: Some methods of our CudaInterval class

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

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 2.2, can be easily implemented on the GPU side with interval arithmetic using our class, as shows listing 3. However, when an interval failure occurs during the sign evaluation, the responsibility to correctly handle the case is delegated to the CPU. Nonetheless, as shown by [1], and reinforced by our case study (sections 4 and 5) interval failures are rare and they usually do not affect the algorithms' overall performance.

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

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

Listing 3: Orientation predicate on GPU

```
1  struct CudaIntervalVertex {
2      CudaInterval x, y;
3  };
4
5  __device__ int orientation(
6      const CudaIntervalVertex* p,
7      const CudaIntervalVertex* q,
8      const CudaIntervalVertex* r) {
9      return ((q->x - p->x) * (r->y - p->y) -
10             (q->y - p->y) * (r->x - p->x)).sign();
11 }
```

4. FAST RED-BLUE INTERSECTION TESTS

To evaluate the ideas presented in this paper, we have implemented a fast and exact algorithm for detecting red-blue intersection of 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 [25]. 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 [26, 27, 28]. Thanks to its simplicity and uniformity, it can be constructed and processed in parallel. For example, Audet et al. [17] employed a uniform grid on a GPU parallel algorithm for map overlay and Magalhães et al. [18] 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. [18], 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. [18], 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 [25]. Figure 2 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 pro-

cessed 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. I.e., 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 3) 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 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 5, 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.

5. EXPERIMENTS

To evaluate the ideas proposed on this paper, 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.

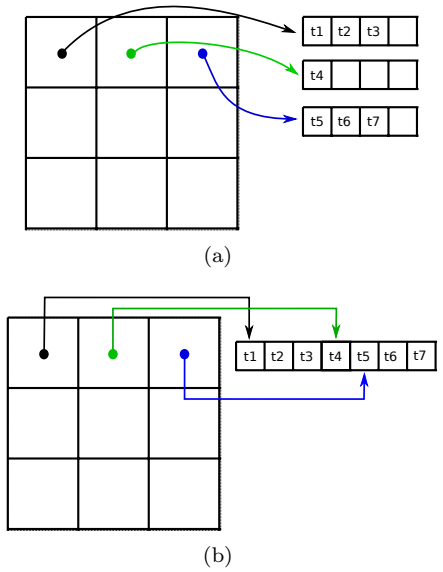


Figure 2: Dynamic array versus ragged array - 3×3 uniform grid using dynamic arrays (a) versus ragged array (b). Only the memory related to the first row of the grid is shown. Source: [25]

Arbitrary-precision arithmetic was provided by the GMP library [15] 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 basing on statistics about the input datasets [25, 29]. 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. [25], 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) 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° (counterclockwise) 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 3 illustrates four of the datasets and Table 5 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 5 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*.
- 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 5 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 (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 paper.

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%) evaluated 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%).

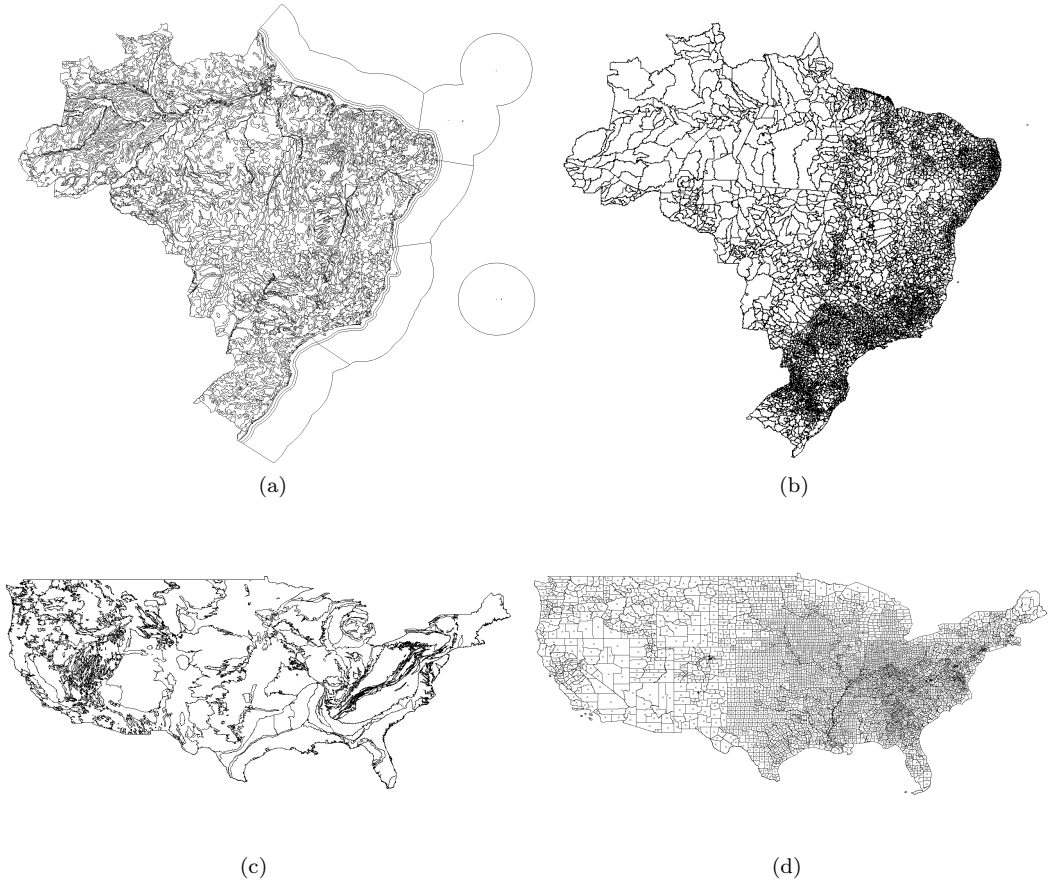
6. CONCLUSIONS AND FUTURE WORK

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

As a proof of concept, a parallel algorithm for detecting intersections of red and blue 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

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

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



(a) (b) (c) (d)

Figure 3: Maps employed in the experiments - BrSoil (a), BrCounty (b), UsAquifers (c), UsCounty (d) (these figures are not to scale).

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
Intersection detection	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
# Intersection 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	1.610	0.392	0.164	5
Intersection detection	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
# Intersection 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
Grid construction	14.532	1.422	7.482	2.798	0.454	0.251	6
Intersection detection	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	1.618	40
# Intersection 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*).

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).

7. ACKNOWLEDGEMENT

This research was partially supported by CAPES.

References

[1] Brönnimann H., Burnikel C., Pion S. "Interval arithmetic yields efficient dynamic filters for computational geometry." *Discrete Applied Mathematics*, vol. 109, no. 1-2, 25–47, 2001

[2] European Space Agency. "Ariane 501 inquiry board report.", 2015. URL `http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf`. (Retrieved on 06/15/2015)

[3] Skeel R. "Roundoff error and the Patriot missile." *SIAM News*, vol. 25, no. 4, 11, July 1992

[4] Kettner L., Mehlhorn K., Pion S., Schirra S., Yap C.K. "Classroom Examples of Robustness Problems in Geometric Computations." *Comput. Geom.*, vol. 40, no. 1, 61–78, May 2008

[5] Hobby J.D. "Practical segment intersection with finite precision output." *Comput. Geom.*, vol. 13, no. 4, 199–214, Oct. 1999

[6] de Berg M., Halperin D., Overmars M. "An intersection-sensitive algorithm for snap rounding." *Computational Geometry*, vol. 36, no. 3, 159–165, Apr. 2007

[7] Hershberger J. "Stable snap rounding." *Comput. Geom.*, vol. 46, no. 4, 403–416, May 2013

[8] Belussi A., Migliorini S., Negri M., Pelagatti G. "Snap Rounding with Restore: An Algorithm for Producing Robust Geometric Datasets." *ACM Trans. Spatial Algorithms and Syst.*, vol. 2, no. 1, 1:1–1:36, Mar. 2016

[9] Shewchuk J.R. "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates." *Discret. & Comput. Geom.*, vol. 18, no. 3, 305–363, Oct. 1997

[10] Li C., Pion S., Yap C.K. "Recent progress in exact geometric computation." *The J. Log. Algebr. Program.*, vol. 64, no. 1, 85–111, July 2005

[11] Hoffman C.M. "The Problems of Accuracy and Robustness in Geometric Computation." *Comput.*, vol. 22, no. 3, 31–40, Mar. 1989

[12] Yap C.K. "Towards exact geometric computation." *Comput. Geom.*, vol. 7, no. 12, 3 – 23, Jan. 1997

[13] Pion S., Fabri A. "A generic lazy evaluation scheme for exact geometric computations." *Sci. Comput. Program.*, vol. 76, no. 4, 307 – 323, Apr. 2011

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

[15] Granlund T., the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.0.0 edn., 2014. `http://gmplib.org/` (Retrieved on 10/19/2017)

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

[17] Audet S., Albertsson C., Murase M., Asahara A. "Robust and Efficient Polygon Overlay on Parallel Stream Processors." *Proc. 21st ACM SIGSPATIAL Int. Conf. Advances Geographic Information Systems*, SIGSPATIAL'13, pp. 304–313. ACM, New York, NY, USA, Nov. 2013

[18] Magalhães S.V., Franklin W.R., Andrade M.V. "Fast exact parallel 3D mesh intersection algorithm using only orientation predicates." *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, p. 44. ACM, 2017

[19] Popescu V. *Towards fast and certified multiple-precision librairies*. Ph.D. thesis, Université de Lyon, 2017

[20] Joldes M., Muller J.M., Popescu V., Tucker W. "CAMPARY: CUDA multiple precision arithmetic library and applications." *International Congress on Mathematical Software*, pp. 232–240. Springer, 2016

[21] Lu M., He B., Luo Q. "Supporting extended precision on graphics processors." *Proceedings of the sixth international workshop on data management on new hardware*, pp. 19–26. ACM, 2010

[22] Collange S., Daumas M., Defour D. "Chapter 9 - Interval Arithmetic in CUDA." W. mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pp. 99 – 107. Morgan Kaufmann, Boston, 2012

[23] Collange S., Flórez J., Defour D. "A GPU interval library based on Boost.Interval." *8th Conference on Real Numbers and Computers*, pp. 61–71. 2008

[24] Whitehead N., Fit-Florea A. "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs." *rn (A+ B)*, vol. 21, no. 1, 18749–19424, 2011

[25] Magalhães S.V.G., Franklin W.R. *Exact and parallel intersection of 3d triangular meshes*. Ph.D. thesis, Rensselaer Polytechnic Institute, USA, 2017

[26] Akman V., Franklin W.R., Kankanhalli M., Narayanaswami C. "Geometric Computing and the Uniform Grid Data Technique." *Comput. Aided Des.*, vol. 21, no. 7, 410–420, Sept. 1989

[27] Franklin W.R., Chandrasekhar N., Kankanhalli M., Seshan M., Akman V. "Efficiency of uniform grids for intersection detection on serial and parallel machines." N. Magnenat-Thalmann, D. Thalmann, editors, *New Trends in Computer Graphics (Proc. Computer Graphics Int.'88)*, pp. 288–297. Springer-Verlag, Berlin, Germany, 1988

[28] Hopkins S., Healey R.G. "A Parallel Implementation of Franklin's Uniform Grid Technique for Line Intersection Detection on a Large Transputer Array." K. Brassel, H. Kishimoto, editors, *4th Int. Symp. Spatial Data Handling*, pp. 95–104. Zürich, 23-27 July 1990

[29] Audet S., Albertsson C., Murase M., Asahara A. "Robust and efficient polygon overlay on parallel stream processors." *Proceedings of the 21st*

ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 304–313. ACM, 2013