EMPLOYING GPUS TO ACCELERATE EXACT GEOMETRIC PREDICATES FOR 3D GEOSPATIAL PROCESSING

Marcelo Menezes
Universidade Federal de Viçosa
marcelo.menezes@ufv.br

Salles V. G. de Magalhães
Universidade Federal de Viçosa
salles@ufv.br

Matheus Aguilar
Universidade Federal de Viçosa
matheus.a.aguilar@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Institute
mail@wrfranklin.org

Bruno Coelho
Universidade Federal de Viçosa
bruno.f.coelho@ufv.br

ABSTRACT

This paper 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.

Keywords computational geometry · 3D GIS · exact computation · high-performance computing · GPU · CUDA

1 Introduction

This Spatial Gem 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 [1], 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 paper we will present a technique to circumvent this challenge.

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 [2], for example, allows the user to choose algorithms from these three categories.

The focus of this Gem is to describe a 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 [3]. Furthermore, it is harder to parallelize exact algorithms because of the frequent memory allocations required as the rationals' precisions grow. Finally, implementing 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 1. Details about evaluating other operations using intervals are described in [4].

Listing 1: Implementation of positiveSum

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

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 1). We propose employing GPUs to accelerate this. One advantage is that GPUs are particularly efficient 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 [5] around 2011.

3 Exact and efficient evaluation of predicates

In this section we present the main ideas behind this Gem. Subsection 3.1 gives an overview of the proposed technique. Subsection 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. For a better understanding of our method, we describe in Subsections 3.4 and 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.

3.1 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 3.2. Section 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 paper. In our previous work [1], 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 3.5 explains the methods used to circumvent the problem.

3.2 Implementing Arithmetic Filters on the GPU

As seen in Section 2, the floating point rounding modes need to be adjusted according to the desired computation. We adopt the same idea as employed by [6] and [1], 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.

Listing 2: Class for interval arithmetic abstraction

```
#define INTERVAL_ERROR 2

class CudaInterval {
public:
    __device__ __host__ CudaInterval(const double l, const double u)
        : lb(l), ub(u) {}
    ...
    __device__ CudaInterval operator+(const CudaInterval& v) const {
        return CudaInterval(__dadd_rd(this->lb, v.lb),
            __dadd_ru(this->ub, v.ub));
    }
    ...
```

```
    __device__ int sign() const {
        if (this->lb > 0) // lb > 0 implies ub > 0
            return 1;
        if (this->ub < 0) // ub < 0 implies lb < 0
            return -1;
        if (this->lb == 0 && this->ub == 0)
            return 0;
        // If none of the above conditions is satisfied,
        // the sign of the exact result cannot be inferred
        // from the interval, Thus, a flag is returned
        // to indicate an interval failure.
        return INTERVAL_ERROR;
    }
    ...
private:
    double lb, ub; // Stores the interval's lower and upper bounds
};
```

The encapsulation of interval arithmetic in a separate class allows for a straightforward implementation of geometric predicates. Listing 3 shows an example implementation for the 2D orientation predicate.

Listing 3: 2D orientation predicate example

```
struct CudaIntervalVertex {
    CudaInterval x, y;
};

__device__ int orientation(
    const CudaIntervalVertex* p,
    const CudaIntervalVertex* q,
    const CudaIntervalVertex* r) {
    return ((q->x - p->x) * (r->y - p->y) -
            (q->y - p->y) * (r->x - p->x)).sign();
}
```

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 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 2 and 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.

3.4 Case study: 2D red/blue segment intersection

As a case study, in [1] 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 [1]. 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 [1], in the worst case only 0.0005% of the predicates failed and required re-evaluation.

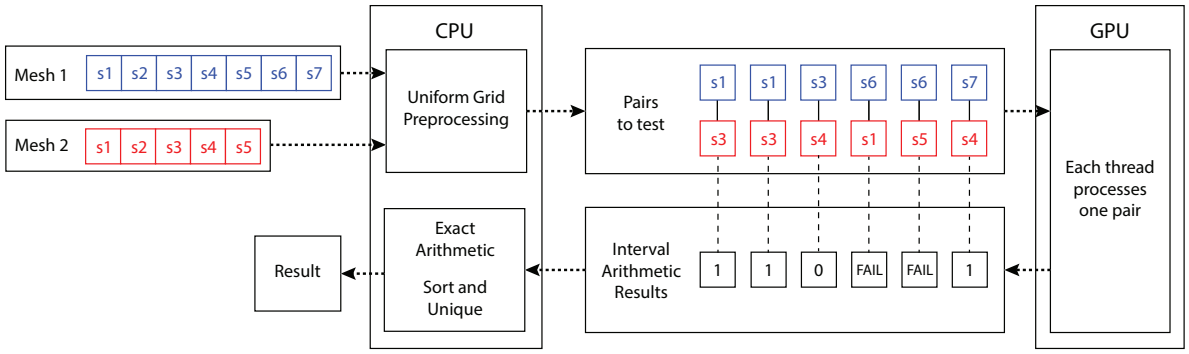

Figure 1: Illustration of the algorithm employed for accelerating exact 2D segment intersections

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 1 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 paper 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.

3.5 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 paper 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 [7].

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 2. 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.

We initially implemented a version of this algorithm using a strategy similar to the one described in Section 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, $\lceil \frac{T_c \times S_c}{Th} \rceil$ 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 paper we assume all indices are 0-based).

Tables 1 and 2 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 2 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 1: 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 2: 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.

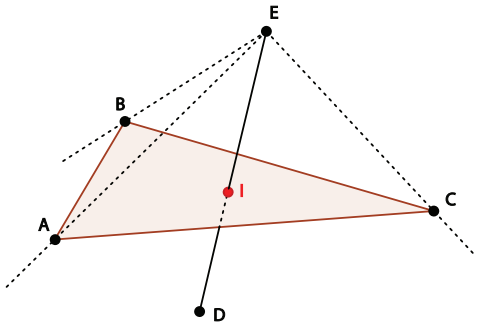

Figure 2: Line segment—triangle intersection test; decomposed into five 3D orientation predicates.

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.

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 [7], 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 3 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 a fast pre-processing with a small intersection detection time on the GPU. Indeed, its intersection step performs two passes (as explained in Section 3.5), 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.

Table 3: 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

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 $25\times$.

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 $77\times$. 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 paper.

5 Conclusion and Future Work

This paper 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 $25\times$ 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 applied 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.

6 Acknowledgements

This research has been partially supported financed by CAPES.

References

[1] Marcelo de Matos Menezes, Salles Viana Gomes Magalhães, W. Randolph Franklin, Matheus Aguilar de Oliveira, and Rodrigo E. O. Bauer Chichorro. Accelerating the exact evaluation of geometric predicates with GPUs. In Suzanne Shontz, Joaquim Peiro, and Ryan Viertel, editors, *28th International Meshing Roundtable*, Buffalo, NY, USA, 16 Oct 2019.

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

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

[4] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1):25 – 47, 2001. 14th European Workshop on Computational Geometry.

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

[6] Sylvain Collange, Marc Daumas, and David Defour. Chapter 9 - interval arithmetic in CUDA. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 99 – 107. Morgan Kaufmann, Boston, 2012.

[7] Lucas C. Villa Real, Bruno Silva, Dikran S. Meliksetian, and Kaique Sacchi. Large-scale 3d geospatial processing made possible. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '19, page 199–208, New York, NY, USA, 2019. Association for Computing Machinery.