Rensselaer Polytechnic Institute, Troy
NY USA
Universidade Federal de Viçosa, MG,
Brazil

RPI

UFV

Accelerating the exact evaluation of geometric predicates with GPUs

Marcelo de Matos Menezes
Salles Viana Gomes Magalhães, UFV/RPI
W. Randolph Franklin, RPI
Matheus Aguilar de Oliveira, UFV
Rodrigo E. O. Bauer Chichorro, UFV

Our Team

- Marcelo de Matos Menezes, UFV: master's student since 2018; interests are Computer Graphics, Computational Geometry and High-Performance Computing, long-term goal to apply the ideas described on this paper to other CG algorithms.
- Salles Viana Gomes Magalhães, RPI PhD grad, UFV prof
- W. Randolph Franklin, RPI prof.
- Matheus Aguilar de Oliveira, UFV: CS undergrad since 2018; interests are Computational Geometry and Competitive Programming.
- Rodrigo E. O. Bauer Chichorro, UFV: CS undergrad since 2017; interests are Competitive Programming, Computational Geometry and Artificial Intelligence





Our research strategy

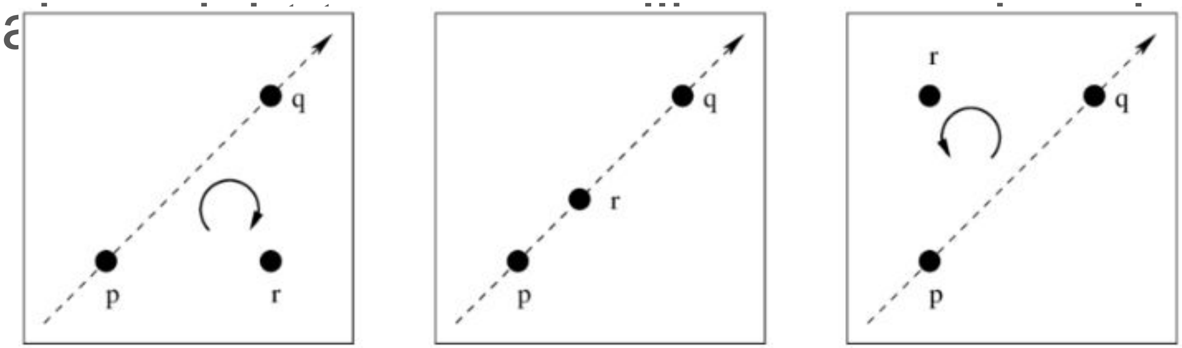
- Identify fundamental geometric operations
- used in higher-level systems
- that need to produce correct results.
- and should execute very fast.
- Devise new theory
- using simple data structures
- on current hardware.
- Implement
- Test.

This paper’s contribution

- A faster solution to erroneous computations caused by floating point finite precision computations.
- Errors can cause predicates (conditionals)to be evaluated wrong.
- That can cause topological errors.
- Existing solutions are either very slow or may fail.
- We synergize three software techniques and two hardware platforms.
  - filter input with uniform grid on CPU, then
  - filter survivors with interval arithmetic on GPU, finally
  - if necessary, compute exactly with multiprecision rationals back on CPU.
- Result: both fast and good.

The problem of roundoff errors

- Floating-point errors: computational geometry challenge.
- Generate topological inconsistencies: global impossibilities.
  - intersection point between two lines may not lie in either.
- Example: planar orientation predicate
  - Do three points p = (px, py), q = (qx, qy) and r = (rx, ry) ma ft turn?

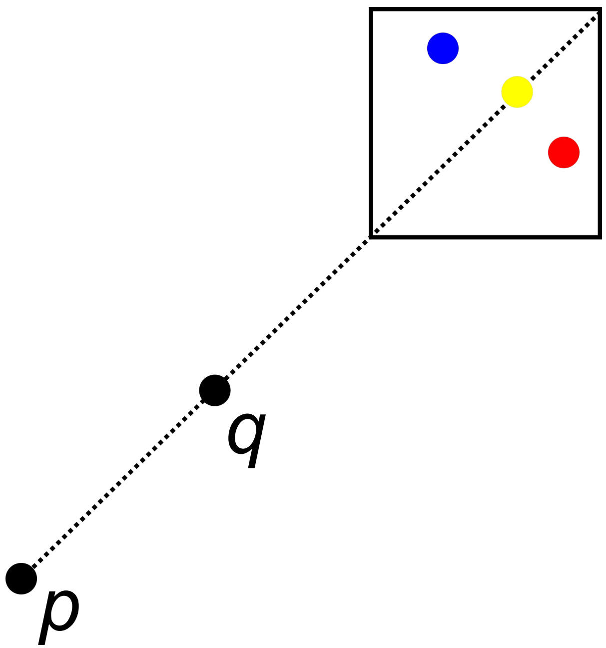


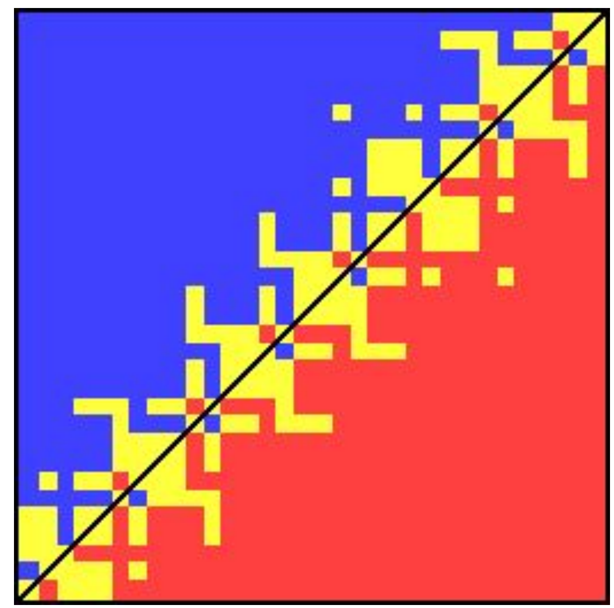
Source: https://www.researchgate.net/figure/The-orientation-predicate-of-3-po n-collinear-and_fig2_1959784

- Predicate = sign of the determinant: $\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix}$

Roundoff errors

- Evaluating the predicate using floating point arithmetic:





Source: Kettner et al., Classroom examples of robustness problems in geometric computations

- Common techniques (snap rounding, epsilon tweaking, etc): no guarantee

Rationals: one roundoff error solution

- Solution for roundoff errors: exact arithmetic (e.g. GMP rationals), but challenges:
  - Slower than floats
  - Size is exponential in depth of computation tree, although that's not a problem if the tree is shallow
  - Growing the size of a variable allocates memory on the global heap.
    - Total time may be superlinear in the number of objects, and
    - is serial,
- Apparently little prior art of working (not just proposed) rational number systems on GPUs

Arithmetic filters and Interval arithmetic (IA), 1

- Technique used in several CG implementations, e.g.: CGAL
- Basic idea: use exact arithmetic only when really necessary
- Predicate evaluation: typically “=” sign of an arithmetic expression
- Each value has:
  - an exact value (can be lazily computed), and
  - an approximation given by an interval [xl, xh].
- Predicates evaluated using the approximation
- If the sign of the exact result can be <u>safely</u> inferred based on results computed with the intervals → use that sign
- Otherwise (a.k.a. <u>filter failure</u>) → re-evaluate with exact arithmetic

Arithmetic filters and IA, 2

- IA used to compute the sign of an expression.
- If it reports a *non-zero* result, it's guaranteed to be correct.
- Sometimes it reports a failure. (Then we escalate.)
- Real *x* represented as $[\underline{x}, \overline{x}]$, where $\underline{x} <= x <= \overline{x}$

$$
\begin{aligned}
[x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
[x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
[x] \cdot [y] &= [\min\{\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy}\}, \max\{\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy}\}] \\
[x]/[y] &= \begin{cases} [x] \cdot [1/\overline{y}, 1/\underline{y}] & \text{if } 0 \notin [y], \\ \mathbb{R} & \text{otherwise} \end{cases} \\
[x]^{1/2} &= \begin{cases} [\underline{x}^{1/2}, \overline{x}^{1/2}] & , \quad \text{if } 0 \leq \underline{x} \\ \mathbb{R} & , \quad \text{otherwise} \end{cases}
\end{aligned}
$$

Source of the fig: Brönnimann, H., Burnikel, C., & Pion, S. (2001). Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, *109*(1-2), 25-47.

Arithmetic filters and IA, 3

- An interval is a pair of floating-point numbers.
- To satisfy the containment property, the operations must change the way floating point values are rounded.
- IEEE-754 standard:
  - Result = exact result rounded to the next (or previous) representable FP number
  - Can be rounded:
    - towards $-\infty$
    - to the closest FP (default)
    - towards $+\infty$
- Changing the rounding mode on a GPU is very fast (slow on CPU)

Arithmetic filters and Interval arithmetic, 4

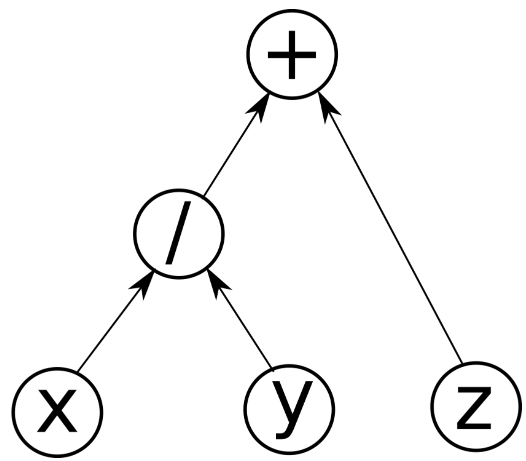
- CGAL uses arithmetic filters/IA- transparent to programmer but not thread safe.
- Illustration of a predicate one could implement:

```
// Predicate: returns true if the sum of x_exact with y_exact is positive
//            and false otherwise. x_interval and y_interval must contain,
//            respectively, x_exact and y_exact.

bool predicate(mpq_class x_exact, CGAL::Interval_nt<> x_interval,
               mpq_class y_exact, CGAL::Interval_nt<> y_interval) {
    try {
        if (x_interval + y_interval > 0)
            return true;
        else
            return false;
    }
    catch (CGAL::Interval_nt<>::unsafe_comparison& ex) {
        if (x_exact + y_exact > 0)
            return true;
        else
            return false;
    }
}
```

Arithmetic filters and Interval arithmetic, 5

- CGAL: arithmetic filtering can be performed “dynamically/automatically”
- Example:
  - A DAG may be created to keep track of results
  - If exact evaluation necessary → lazily re-evaluate the values

```
a = x/y
b = a + z
if(b>0) {
  ….
}
```



Exact fast parallel intersection of large 3-D triangular meshes

- Earlier work, presented last year
- Salles Magalhaes thesis
- Intersected 3D meshes using shared-memory multi-core CPUs. Combined:
  - Simulation of Simplicity
  - Arithmetic filtering/IA. “Manually” managed.
  - Parallel on multicore Intel Xeon with OpenMP
  - Big rationals.
- Today: start to incorporate GPUs.

Idea for using exact computation and GPUs

- GPUs:
  - excellent for floating-point arithmetic
  - however, warps of 32 threads should run same instruction stream on adjacent data
  - trees, hierarchical data structures, pointers are very inefficient.
- Implement the IA computation on the GPU
- CPU batch offloads evaluation of predicates to GPU.
- Indeterminate results are filtered and re-evaluated on the CPU.

What is CUDA?

- To program Nvidia GPUs.
- C++ with small syntax extensions and library.
- nvcc compiler separates program into code for CPU host and code for GPU device.
- GPU architecture is complicated.
    - thousands of cores, each 1/20 as powerful as Xeon core
    - SIMT, 32 thread warp
    - several memory classes:
        - varying speed,
        - size (to 48GB),
        - latency,
        - unified VM with host.
- A range of higher level abstract layers like Thrust and Kokkos trade off programmer time and execution time.

Implementation details, 1

- Created a class, based on Collange et al., to perform the necessary calculations → easier usage
- The rounding modes on CUDA C are selected via compiler intrinsics:
  - e.g.: For addition:
    - __dadd_rd() switches the rounding mode towards $-\infty$
    - __dadd_ru() switches the rounding mode towards $+\infty$
- These are hidden from the user through operator overloading

Implementation details, 2

Some methods in our *CudaInterval* class

```
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 2;
    }
    ...
private:
    double lb, ub; // Stores the interval's lower and upper bounds
};
```

Implementation details, 3

- Predicates: easily implemented using class instances
- Example: 2D orientation predicate

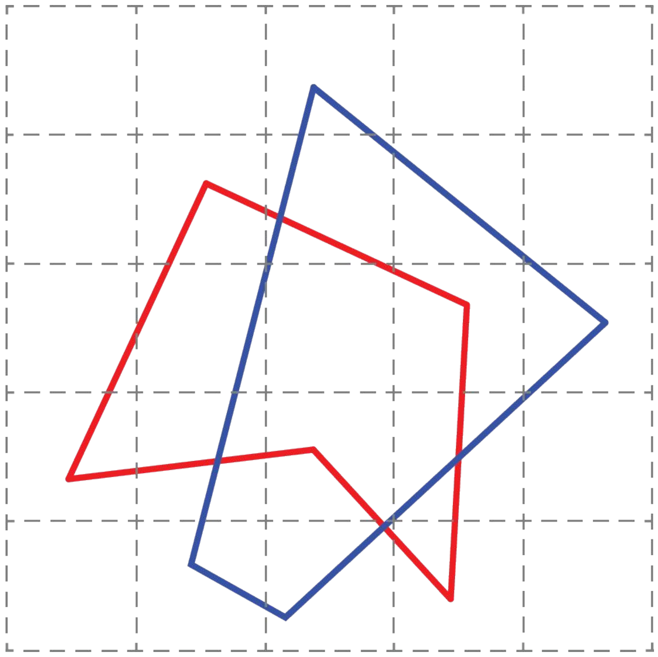
```
struct CudaIntervalVertex {
    CudaInterval x, y;
};

__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();
}
```

Fast red-blue intersection tests

- Case study: fast and exact algorithm for detecting red-blue intersection of line segments.
- Given two sets of segments *S1* (red segments) and *S2* (blue segments) $\rightarrow$ find pairs of red-blue intersections.
- Possible quadratic number of red-red and blue-blue intersections, even though few red-blue intersections.
- So, harder than finding all segment intersections.
  - Sweep line is too inefficient here.
- Algorithm steps:
  - Uniform grid preprocessing filter on CPU identifies pairs of segments that may intersect
  - Interval analysis tests further filters those pairs on GPU,
  - Exact rational arithmetic back on CPU exactly tests a few pairs.

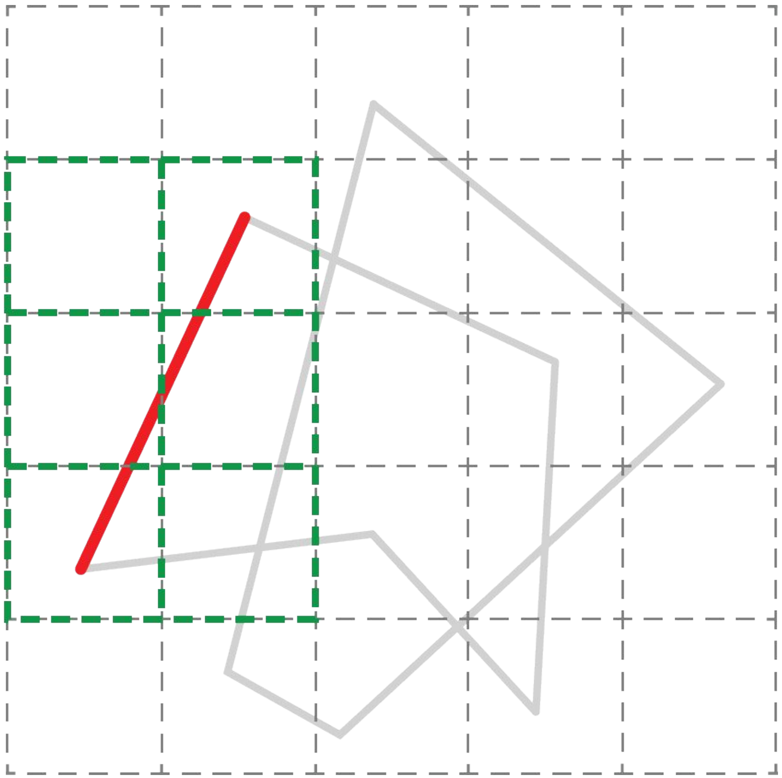
Fast red-blue intersections: Pre-processing, 1

- Consider the following segment sets *S1* (red) and S2 (blue):
- A uniform grid divides the domain into equally sized regions:



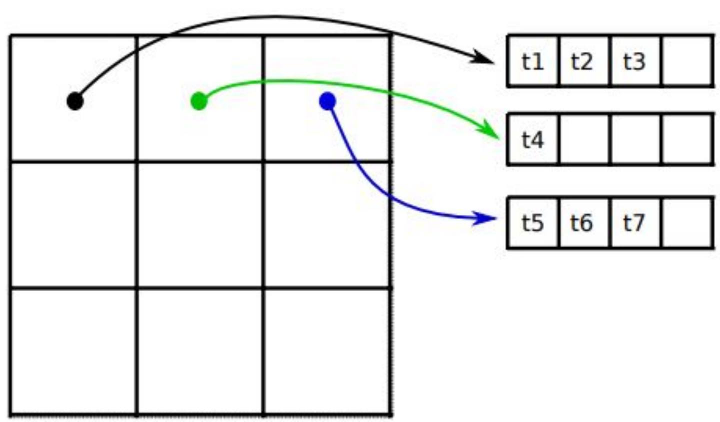
Fast red-blue intersections: Pre-processing, 2

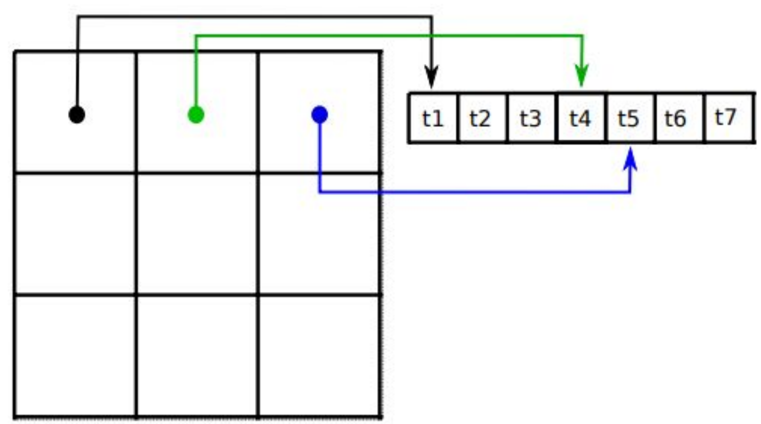
- Each segment (from both sets) is associated with the grid cells its bounding box intercepts.
- (Possible future mod would compute exactly which cells intersect the segment.)



Fast red-blue intersections: Pre-processing, 3

- For time and space efficiency, use a ragged array
  - One array containing all the elements, plus
  - Dope vector pointing to start of each cell's contents.
  - Constant time to read cell #i element #j.
- Creation requires two passes:
  - Count the number of elements in each cell, then
  - Insert the edges into the ragged array
- Both passes parallelize - faster than dynamic sized arrays



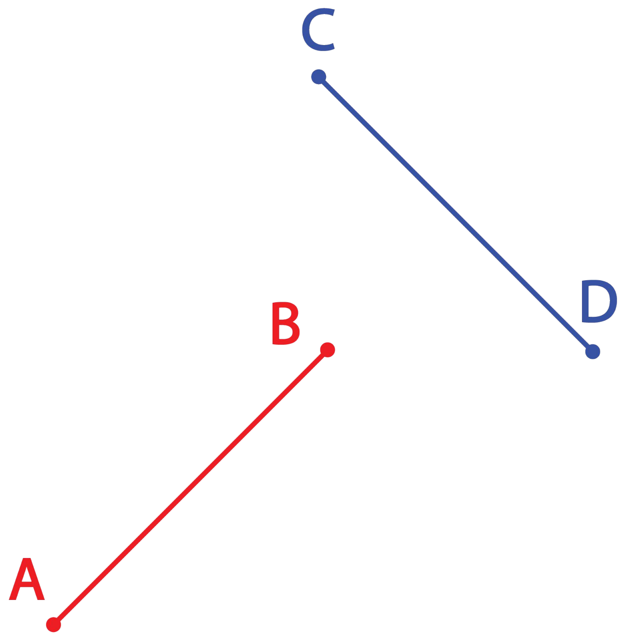


Fast red-blue intersections: Pre-processing, 4

- Once the uniform grid is constructed, a list 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
  - first pass to perform the count of pairs of segments
  - second pass to insert the pairs into the list
- The list can than be sent to the GPU, which will evaluate which of those pairs do intersect.

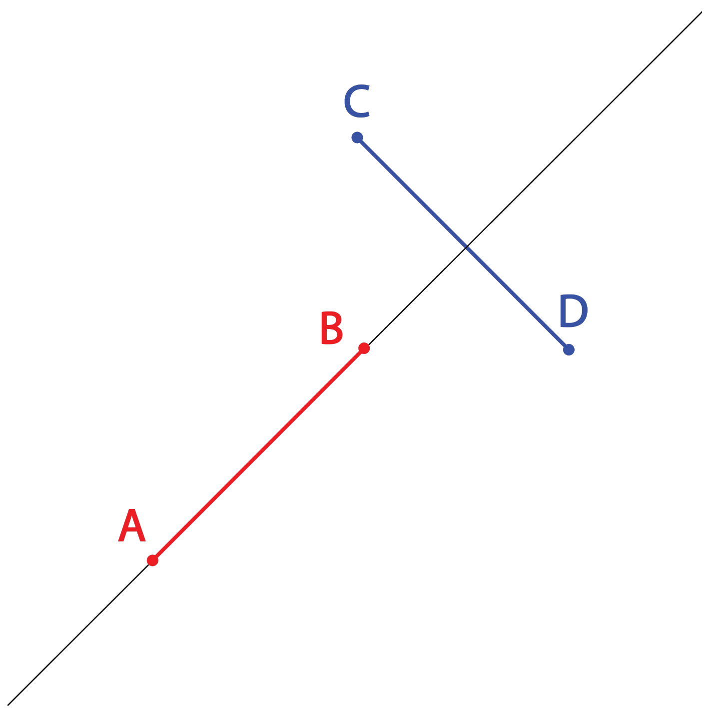
Intersection testing, 1

- Consider the segments AB and CD pictured below
- Four orientation predicates are sufficient to determine if they intersect or not
- **intersect((A,B) , (C,D)) = orientation(A, B, C) $\neq$ orientation(A, B, D) $\wedge$ orientation(C, D, A) $\neq$ orientation(C, D, B)**

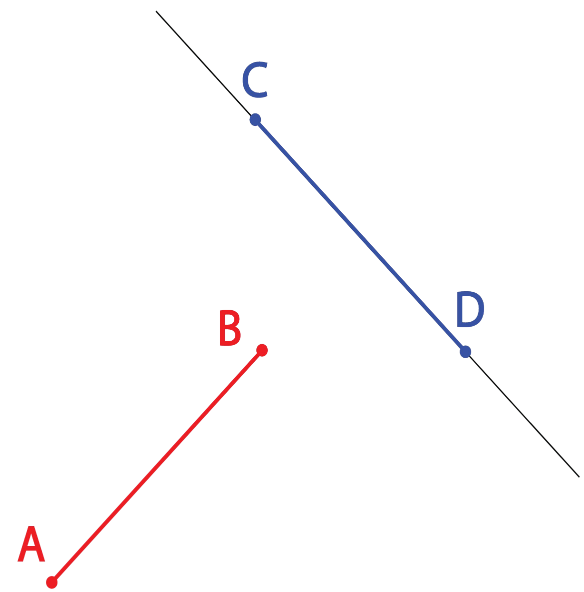


Intersection testing, 2

- C and D have different orientations w.r.t. (A,B)
- → CD **intersects** the supporting line of AB



- A and B have the same orientation w.r.t. (C,D)
- → AB **does not intersect** the supporting line of CD



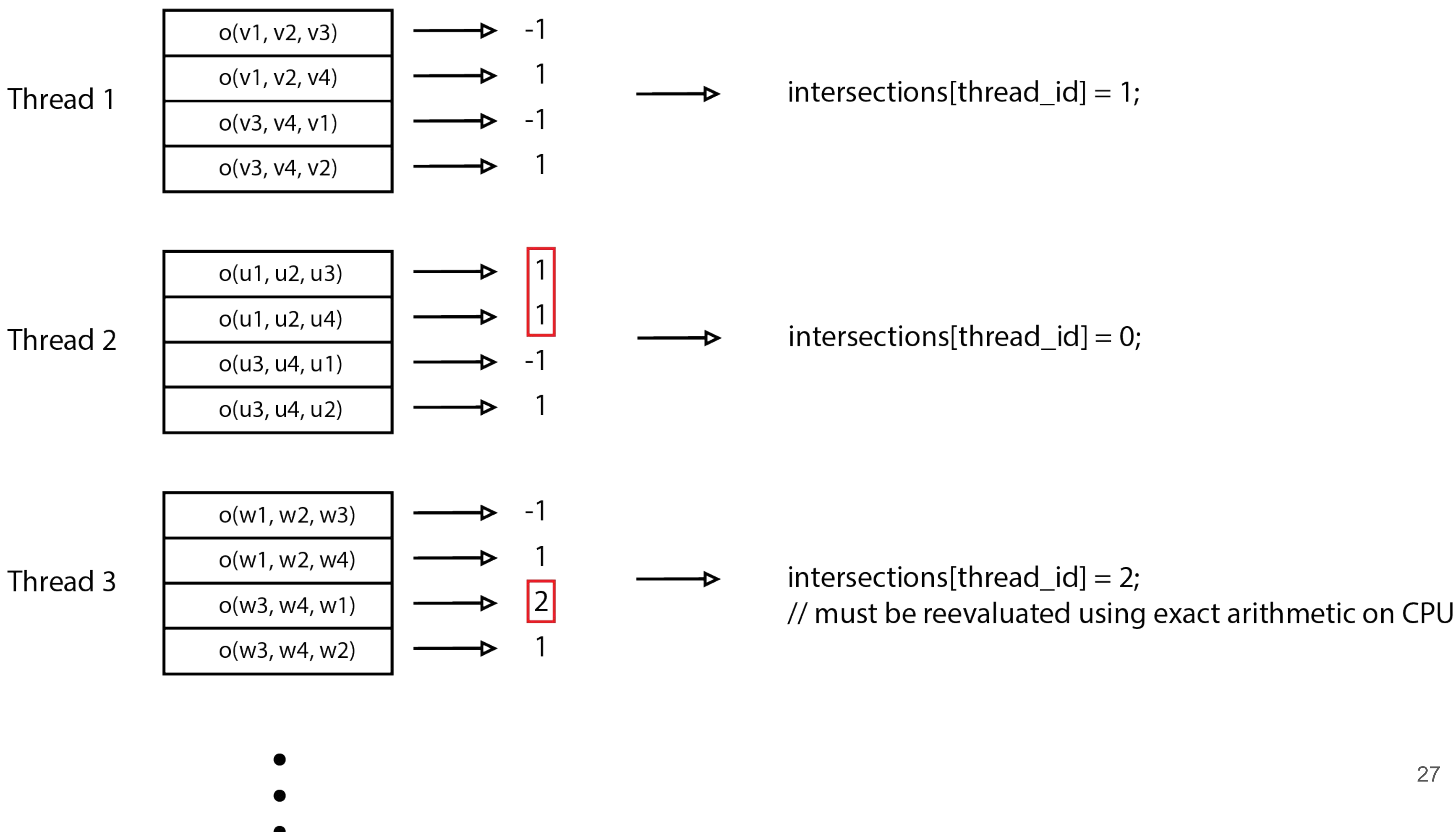
Intersection testing, 3

- A CPU implementation checks one pair of segments at a time, evaluating the four predicates in a for loop:

```
...
for (int i = 0; i < n; ++i) {
    Vertex v1 = set_1_edges[i].v1;
    Vertex v2 = set_1_edges[i].v2;
    Vertex v3 = set_2_edges[i].v1;
    Vertex v4 = set_2_edges[i].v2;

    int o1 = orientation(v1, v2, v3);
    int o2 = orientation(v1, v2, v4);
    int o3 = orientation(v3, v4, v1);
    int o4 = orientation(v3, v4, v2);

    intersections[i] = (o1 != o2) && (o3 != o4);
}
...
```

Intersection testing, 4

- List of pairs sent in one batch to GPU.
- One thread does one intersection test.



Experiments, 1

- Environment:
  - AMD Ryzen 5 processor with 6 3.2GHz cores (12 hyperthreads)
  - 16 GB of RAM
  - NVIDIA GeForce GTX 1070 Ti GPU
- Arbitrary precision arithmetic provided by the GMP library
- OpenMP for parallelizing the CPU code
- Cuda for the GPU side
- Compared against CGAL:
  - Sequential method for detecting intersections of dD Iso-oriented Boxes (pre-processing)
  - Arithmetic filtering and lazy evaluation

Experiments, 2

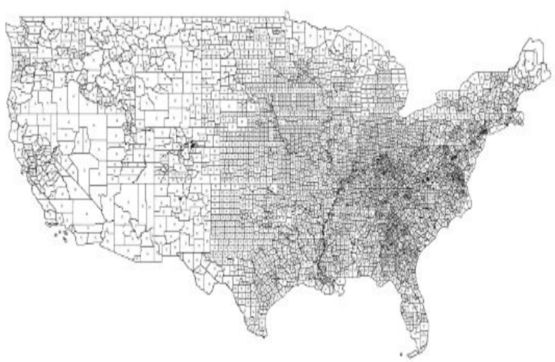
- Experiments have been performed using segments from four polygonal maps from two countries
- The intersection tests were made in pairs, using a 2500x2500 resolution uniform grid:
  - BrSoil x BrCounty
  - UsCounty x UsAquifers
  - UsCounty x UsCountyRotated
- Properties of each map:

	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

Experiments, 3

BrSoil and BrCounty

UsCounty and Us Aquifers



Notes on the test data

- The edge segments are very unevenly distributed.
- Most grid cells are empty, a few have many edges.
- Yet the uniform grid works well.
- Quadtrees etc are not necessary (and are slower and don’t parallelize well).
- Most intersection tests fail.
- That’s ok because they’re very fast and parallelize.

no filtering

filtering, lazy evaluation...

vs. Interval*

BrCounty and BrSoil							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	1.242	0.225	0.478	0.549	0.324	0.099	2
Inter.	1.444	0.152	0.015	0.385	0.040	0.018	9
Total	2.686	0.377	0.493	0.934	0.364	0.117	3
# tests	300K	300K	70K	300K	300K	300K	-

UsCounty and UsAquifers							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	7.884	0.812	2.628	1.610	0.392	0.164	5
Inter.	42.816	4.059	0.023	11.198	0.612	0.096	42
Total	50.700	4.871	2.651	12.808	1.004	0.260	19
# tests	13M	13M	159K	13M	13M	13M	-

UsCounty and UsCountyRotated							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	14.532	1.422	7.482	2.798	0.454	0.251	6
Inter.	675.616	63.677	1.027	194.918	9.422	1.367	47
Total	690.148	65.099	8.509	197.716	9.876	1.618	40
# tests	217M	217M	11M	217M	217M	217M	-

Parallel

BrCounty and BrSoil	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	1.242	0.225	0.478	0.549	0.324	0.099	2
Inter.	1.444	0.152	0.015	0.385	0.040	0.018	9
Total	2.686	0.377	0.493	0.934	0.364	0.117	3
# tests	300K	300K	70K	300K	300K	300K	-

UsCounty and UsAquifers	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	7.884	0.812	2.628	1.610	0.392	0.164	5
Inter.	42.816	4.059	0.023	11.198	0.612	0.096	42
Total	50.700	4.871	2.651	12.808	1.004	0.260	19
# tests	13M	13M	159K	13M	13M	13M	-

UsCounty and UsCountyRotated	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	14.532	1.422	7.482	2.798	0.454	0.251	6
Inter.	675.616	63.677	1.027	194.918	9.422	1.367	47
Total	690.148	65.099	8.509	197.716	9.876	1.618	40
# tests	217M	217M	11M	217M	217M	217M	-

CGAL: better pre-processing culling (but slower)

Interval*: faster culling and can be parallelized

Time not exactly proportional to number of tests (faster if pair does not intersect)

BrCounty and BrSoil							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	1.242	0.225	0.478	0.549	0.324	0.099	2
Inter.	1.444	0.152	0.015	0.385	0.040	0.018	9
Total	2.686	0.377	0.493	0.934	0.364	0.117	3
# tests	300K	300K	70K	300K	300K	300K	-

UsCounty and UsAquifers							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	7.884	0.812	2.628	1.610	0.392	0.164	5
Inter.	42.816	4.059	0.023	11.198	0.612	0.096	42
Total	50.700	4.871	2.651	12.808	1.004	0.260	19
# tests	13M	13M	159K	13M	13M	13M	-

UsCounty and UsCountyRotated							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	14.532	1.422	7.482	2.798	0.454	0.251	6
Inter.	675.616	63.677	1.027	194.918	9.422	1.367	47
Total	690.148	65.099	8.509	197.716	9.876	1.618	40
# tests	217M	217M	11M	217M	217M	217M	-

Effect of arithmetic filtering

Filters failed in only 0.000002% to 0.0005% of the predicates

→ Rationals rarely necessary

→ In the GPU implementation, CPU rarely had to re-evaluate with rationals.

	BrCounty and BrSoil						
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	1.242	0.225	0.478	0.549	0.324	0.099	2
Inter.	1.444	0.152	0.015	0.385	0.040	0.018	9
Total	2.686	0.377	0.493	0.934	0.364	0.117	3
# tests	300K	300K	70K	300K	300K	300K	-

	UsCounty and UsAquifers						
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	7.884	0.812	2.628	1.610	0.392	0.164	5
Inter.	42.816	4.059	0.023	11.198	0.612	0.096	42
Total	50.700	4.871	2.651	12.808	1.004	0.260	19
# tests	13M	13M	159K	13M	13M	13M	-

	UsCounty and UsCountyRotated						
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	14.532	1.422	7.482	2.798	0.454	0.251	6
Inter.	675.616	63.677	1.027	194.918	9.422	1.367	47
Total	690.148	65.099	8.509	197.716	9.876	1.618	40
# tests	217M	217M	11M	217M	217M	217M	-

Pre-processing: not entirely on the CPU -- GPU computes in which cell each vertex is.
(this is not a predicate, but can be computed with IA and filtering)

GPU pre-processing: includes copying intervals (coordinates) to the GPU.

BrCounty and BrSoil							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	1.242	0.225	0.478	0.549	0.324	0.099	2
Inter.	1.444	0.152	0.015	0.385	0.040	0.018	9
Total	2.686	0.377	0.493	0.934	0.364	0.117	3
# tests	300K	300K	70K	300K	300K	300K	-

UsCounty and UsAquifers							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	7.884	0.812	2.628	1.610	0.392	0.164	5
Inter.	42.816	4.059	0.023	11.198	0.612	0.096	42
Total	50.700	4.871	2.651	12.808	1.004	0.260	19
# tests	13M	13M	159K	13M	13M	13M	-

UsCounty and UsCountyRotated							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	14.532	1.422	7.482	2.798	0.454	0.251	6
Inter.	675.616	63.677	1.027	194.918	9.422	1.367	47
Total	690.148	65.099	8.509	197.716	9.876	1.618	40
# tests	217M	217M	11M	217M	217M	217M	-

Inter.: 0.685s prepare + 62.992s eval. (prep. = generate (parallel) list of edges to test)

GPU : 1.149s prepare + 0.218s eval. (prep. = same as CPU + copy ids to/from GPU)

289x speedup in evaluation→Possibly better speedups in algs. w/less communication

Conclusions and future work, 1

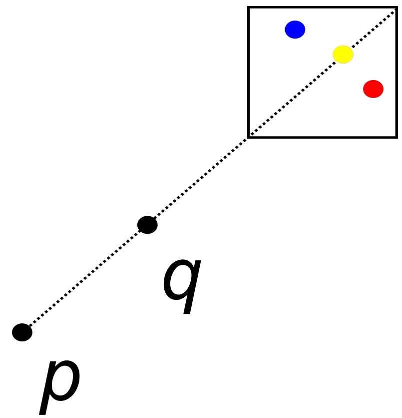
- Good for interactive applications (CAD, GIS, CG, ...)
- Intervals do not fail often (fail → re-evaluation)
- More efficient to keep data on the GPU and re-use
    - If coordinates will be re-used, copy to the GPU at the beginning of the program.
    - Use communication only for what is really necessary. (e.g.: for intersections, copy the ids of the pairs of the edges)
    - E.g.: boolean operations: detecting intersection is only one step → data can be kept on the GPU and re-used in all steps.

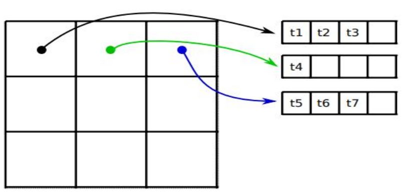
Conclusions and future work, 2

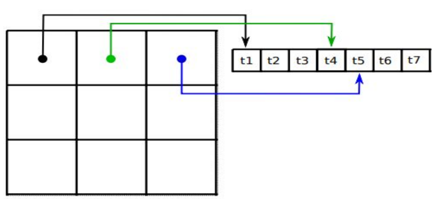
- Future work:
  - Apply to 2D/3D point location, mesh intersection and other CG algorithms.
  - Improve the performance of the predicates.
    - E.g.: reduce CPU-GPU communication overhead (move combinatorial part of the algorithm to GPU, overlap communication/processing, etc).
- Challenges:
  - Predicates must be evaluated in batch
  - Have to “manually” keep track of how each interval was generated (ok mainly when depth of the computation tree is small)
  - Intervals may fail more often in applications with deep computation trees.

Mesh intersection

Thank you!



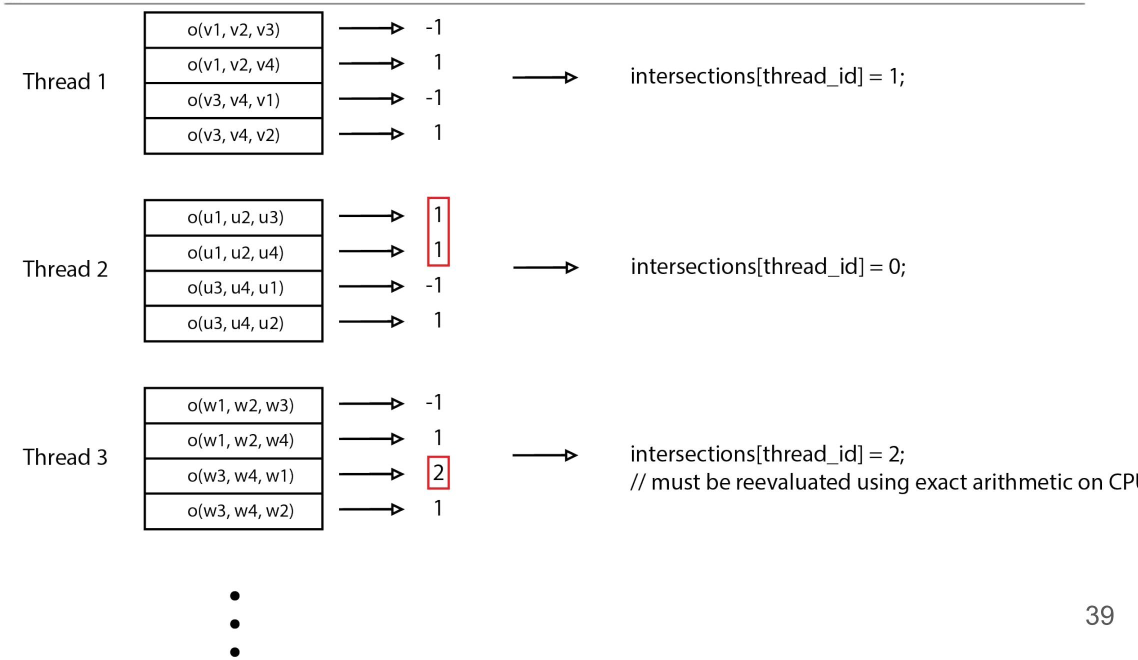




BrCounty and BrSoil							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	1.242	0.225	0.478	0.549	0.324	0.099	2
Inter.	1.444	0.152	0.015	0.385	0.040	0.018	9
Total	2.686	0.377	0.493	0.934	0.364	0.117	3
# tests	300K	300K	70K	300K	300K	300K	-

UsCounty and UsAquifers							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	7.884	0.812	2.628	1.610	0.392	0.164	5
Inter.	42.816	4.059	0.023	11.198	0.612	0.096	42
Total	50.700	4.871	2.651	12.808	1.004	0.260	19
# tests	13M	13M	159K	13M	13M	13M	-

UsCounty and UsCountyRotated							
	Rational*	Interval*	CGAL*	Rational	Interval	GPU	Speedup
Pre.	14.532	1.422	7.482	2.798	0.454	0.251	6
Inter.	675.616	63.677	1.027	194.918	9.422	1.367	47
Total	690.148	65.099	8.509	197.716	9.876	1.618	40
# tests	217M	217M	11M	217M	217M	217M	-



Acknowledgement: CAPES CNPq