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 $r = (rx, ry)$ make a right turn, are collinear or make a left turn?

Predicate = sign of the determinant:

$$\begin{vmatrix}
px & py & 1 \\
qx & qy & 1 \\
rx & ry & 1
\end{vmatrix}$$

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

r-collinear-and_fig2_1959784
Roundoff errors

- Evaluating the predicate using floating point arithmetic:

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

Source: Kettner et al., Classroom examples of robustness problems in geometric computations
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 $[x_l, x_h]$. 
- Predicates evaluated using the approximation
- If the sign of the exact result can be safely inferred based on results computed with the intervals $\rightarrow$ use that sign
- Otherwise (a.k.a. filter failure) $\rightarrow$ 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 $[x, \bar{x}]$, where $x \leq x \leq \bar{x}$

\[
[x] + [y] = [x + y, \bar{x} + \bar{y}]
\]
\[
[x] - [y] = [x - \bar{y}, \bar{x} - y]
\]
\[
[x] \cdot [y] = [\min\{xy, x\bar{y}, \bar{x}y, \bar{x}\bar{y}\}, \max\{xy, x\bar{y}, \bar{x}y, \bar{x}\bar{y}\}]
\]
\[
[x]/[y] = \begin{cases} [x] \cdot [1/\bar{y}, 1/y] & \text{if } 0 \not\in [y], \\ \mathbb{R} & \text{otherwise} \end{cases}
\]
\[
[x]^{1/2} = \begin{cases} \left[\frac{x^{1/2}}, \frac{1}{\bar{x}^{1/2}}\right] & \text{if } 0 \leq x \\ \mathbb{R} & \text{otherwise} \end{cases}
\]

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:

```cpp
// 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

```cpp
class CudaInterval {
public:
  __device__ __host__ CudaInterval(const double l, const double u) 
    : lb(l), ub(u) {}
...
  __device__ CudaInterval+(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

```cpp
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 }
```
Fast red-blue intersection tests

- Case study: fast and exact algorithm for detecting red-blue intersection of line segments.
- Given two sets of segments $S_1$ (red segments) and $S_2$ (blue segments) → 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:
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 then 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.
- \( \text{intersect}( (A, B), (C, D) ) = \text{orientation}(A, B, C) \neq \text{orientation}(A, B, D) \land \text{orientation}(C, D, A) \neq \text{orientation}(C, D, B) \)
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:

```java
1 ... 
2     for (int i = 0; i < n; ++i) {
3         Vertex v1 = set_1_edges[i].v1;
4         Vertex v2 = set_1_edges[i].v2;
5         Vertex v3 = set_2_edges[i].v1;
6         Vertex v4 = set_2_edges[i].v2;
7     
8         int o1 = orientation(v1, v2, v3);
9         int o2 = orientation(v1, v2, v4);
10        int o3 = orientation(v3, v4, v1);
11        int o4 = orientation(v3, v4, v2);
12        
13        intersections[i] = (o1 != o2) && (o3 != o4);
14     }
15     ...
```
Intersection testing, 4

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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o(v1, v2, v3)</td>
<td>→ -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(v1, v2, v4)</td>
<td>→ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(v3, v4, v1)</td>
<td>→ -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(v3, v4, v2)</td>
<td>→ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{intersections[thread_id]} = 1; \]

<table>
<thead>
<tr>
<th>Thread 2</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o(u1, u2, u3)</td>
<td>→ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(u1, u2, u4)</td>
<td>→ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(u3, u4, u1)</td>
<td>→ -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(u3, u4, u2)</td>
<td>→ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{intersections[thread_id]} = 0; \]

<table>
<thead>
<tr>
<th>Thread 3</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o(w1, w2, w3)</td>
<td>→ -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(w1, w2, w4)</td>
<td>→ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(w3, w4, w1)</td>
<td>→ 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o(w3, w4, w2)</td>
<td>→ 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{intersections[thread_id]} = 2; \]

// must be reevaluated using exact arithmetic on CPU
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:

<table>
<thead>
<tr>
<th></th>
<th>BrSoil</th>
<th>BrCounty</th>
<th>UsCounty</th>
<th>UsAquifers</th>
<th>UsCounty</th>
<th>UsCountyRot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of segments</td>
<td>211,011</td>
<td>326,193</td>
<td>3,740,989</td>
<td>352,924</td>
<td>3,740,989</td>
<td>3,740,989</td>
</tr>
<tr>
<td>Average segment length</td>
<td>$5 \times 10^{-4}$</td>
<td>$4 \times 10^{-4}$</td>
<td>$8 \times 10^{-7}$</td>
<td>$1 \times 10^{-4}$</td>
<td>$8 \times 10^{-7}$</td>
<td>$8 \times 10^{-7}$</td>
</tr>
<tr>
<td>Percentage of empty grid cells</td>
<td>86%</td>
<td></td>
<td>98%</td>
<td></td>
<td>98%</td>
<td></td>
</tr>
<tr>
<td>Average # pairs of segments/cell</td>
<td>0.3</td>
<td></td>
<td>2.0</td>
<td></td>
<td>34.7</td>
<td></td>
</tr>
<tr>
<td>Number of pairs of segments</td>
<td>300,039</td>
<td></td>
<td>12,756,283</td>
<td>11,948</td>
<td>216,542,974</td>
<td>11,751</td>
</tr>
<tr>
<td>Number of intersections</td>
<td>20,860</td>
<td></td>
<td>11,948</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
## Parallel vs. Interval*

<table>
<thead>
<tr>
<th></th>
<th>BrCounty and BrSoil</th>
<th></th>
<th></th>
<th>UsCounty and UsAquifers</th>
<th></th>
<th></th>
<th>UsCounty and UsCountyRotated</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rational*</td>
<td>Interval*</td>
<td>CGAL*</td>
<td>Rational</td>
<td>Interval</td>
<td>GPU</td>
<td>Speedup</td>
<td></td>
</tr>
<tr>
<td>Pre.</td>
<td>1.242</td>
<td>0.225</td>
<td>0.478</td>
<td>0.549</td>
<td>0.324</td>
<td>0.099</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Inter.</td>
<td>1.444</td>
<td>0.152</td>
<td>0.015</td>
<td>0.385</td>
<td>0.040</td>
<td>0.018</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2.686</td>
<td>0.377</td>
<td>0.493</td>
<td>0.934</td>
<td>0.364</td>
<td>0.117</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td># tests</td>
<td>300K</td>
<td>300K</td>
<td>70K</td>
<td>300K</td>
<td>300K</td>
<td>300K</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

---

**Note:**
- **no filtering**
- **filtering, lazy evaluation...**

---

Parallel
<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrCounty and BrSoil</td>
<td>Pre.</td>
<td>1.242</td>
<td>0.225</td>
<td>0.478</td>
<td>0.549</td>
<td>0.324</td>
<td>0.099</td>
</tr>
<tr>
<td></td>
<td>Inter.</td>
<td>1.444</td>
<td>0.152</td>
<td>0.015</td>
<td>0.385</td>
<td>0.040</td>
<td>0.018</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>2.686</td>
<td>0.377</td>
<td>0.493</td>
<td>0.934</td>
<td>0.364</td>
<td>0.117</td>
</tr>
<tr>
<td></td>
<td># tests</td>
<td>300K</td>
<td>300K</td>
<td>70K</td>
<td>300K</td>
<td>300K</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>UsCounty and UsAquifers</td>
<td>Pre.</td>
<td>7.884</td>
<td>0.812</td>
<td>2.628</td>
<td>1.610</td>
<td>0.392</td>
<td>0.164</td>
</tr>
<tr>
<td></td>
<td>Inter.</td>
<td>42.816</td>
<td>4.059</td>
<td>0.023</td>
<td>11.198</td>
<td>0.612</td>
<td>0.096</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>50.700</td>
<td>4.871</td>
<td>2.651</td>
<td>12.808</td>
<td>1.004</td>
<td>0.260</td>
</tr>
<tr>
<td></td>
<td># tests</td>
<td>13M</td>
<td>13M</td>
<td>159K</td>
<td>13M</td>
<td>13M</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>UsCounty and UsCountyRotated</td>
<td>Pre.</td>
<td>14.532</td>
<td>1.422</td>
<td>7.482</td>
<td>2.798</td>
<td>0.454</td>
<td>0.251</td>
</tr>
<tr>
<td></td>
<td>Inter.</td>
<td>675.616</td>
<td>63.677</td>
<td>1.027</td>
<td>194.918</td>
<td>9.422</td>
<td>1.367</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>690.148</td>
<td>65.099</td>
<td>8.509</td>
<td>197.716</td>
<td>9.876</td>
<td>1.618</td>
</tr>
<tr>
<td></td>
<td># tests</td>
<td>217M</td>
<td>217M</td>
<td>11M</td>
<td>217M</td>
<td>217M</td>
<td>-</td>
</tr>
</tbody>
</table>

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)
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.
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.

<table>
<thead>
<tr>
<th></th>
<th>BrCounty and BrSoil</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rational*</td>
<td>Interval*</td>
<td>CGAL*</td>
<td>Rational</td>
<td>Interval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre.</td>
<td>1.242</td>
<td>0.225</td>
<td>0.478</td>
<td>0.549</td>
<td>0.324</td>
<td>0.099</td>
<td>2</td>
</tr>
<tr>
<td>Inter.</td>
<td>1.444</td>
<td>0.152</td>
<td>0.015</td>
<td>0.385</td>
<td>0.040</td>
<td>0.018</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>2.686</td>
<td>0.377</td>
<td>0.493</td>
<td>0.934</td>
<td>0.364</td>
<td>0.117</td>
<td>3</td>
</tr>
<tr>
<td># tests</td>
<td>300K</td>
<td>300K</td>
<td>70K</td>
<td>300K</td>
<td>300K</td>
<td>300K</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>UsCounty and UsCountyRotated</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rational*</td>
<td>Interval*</td>
<td>CGAL*</td>
<td>Rational</td>
<td>Interval</td>
<td>GPU</td>
<td>Speedup</td>
</tr>
<tr>
<td>Pre.</td>
<td>14.532</td>
<td>1.422</td>
<td>7.482</td>
<td>2.798</td>
<td>0.454</td>
<td>0.251</td>
<td>6</td>
</tr>
<tr>
<td>Inter.</td>
<td>675.616</td>
<td>63.677</td>
<td>1.027</td>
<td>194.918</td>
<td>9.422</td>
<td>1.367</td>
<td>47</td>
</tr>
<tr>
<td>Total</td>
<td>690.148</td>
<td>65.099</td>
<td>8.509</td>
<td>197.716</td>
<td>9.876</td>
<td>1.618</td>
<td>40</td>
</tr>
<tr>
<td># tests</td>
<td>217M</td>
<td>217M</td>
<td>11M</td>
<td>217M</td>
<td>217M</td>
<td>217M</td>
<td></td>
</tr>
</tbody>
</table>

35
### BrCounty and BrSoil

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre.</td>
<td>1.242</td>
<td>0.225</td>
<td>0.478</td>
<td>0.549</td>
<td>0.324</td>
<td>0.099</td>
<td>2</td>
</tr>
<tr>
<td>Inter.</td>
<td>1.444</td>
<td>0.152</td>
<td>0.015</td>
<td>0.385</td>
<td>0.040</td>
<td>0.018</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>2.686</td>
<td>0.377</td>
<td>0.493</td>
<td>0.934</td>
<td>0.364</td>
<td>0.117</td>
<td>3</td>
</tr>
<tr>
<td># tests</td>
<td>300K</td>
<td>300K</td>
<td>70K</td>
<td>300K</td>
<td>300K</td>
<td>300K</td>
<td>-</td>
</tr>
</tbody>
</table>

### UsCounty and UsAquifers

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre.</td>
<td>7.884</td>
<td>0.812</td>
<td>2.628</td>
<td>1.610</td>
<td>0.392</td>
<td>0.164</td>
<td>5</td>
</tr>
<tr>
<td>Inter.</td>
<td>42.816</td>
<td>4.059</td>
<td>0.023</td>
<td>11.198</td>
<td>0.612</td>
<td>0.096</td>
<td>42</td>
</tr>
<tr>
<td>Total</td>
<td>50.700</td>
<td>4.871</td>
<td>2.651</td>
<td>12.808</td>
<td>1.004</td>
<td>0.260</td>
<td>19</td>
</tr>
<tr>
<td># tests</td>
<td>13M</td>
<td>13M</td>
<td>159K</td>
<td>13M</td>
<td>13M</td>
<td>13M</td>
<td>-</td>
</tr>
</tbody>
</table>

### UsCounty and UsCountyRotated

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre.</td>
<td>14.532</td>
<td>1.422</td>
<td>7.482</td>
<td>2.798</td>
<td>0.454</td>
<td>0.251</td>
<td>6</td>
</tr>
<tr>
<td>Inter.</td>
<td>675.616</td>
<td>63.677</td>
<td>1.027</td>
<td>194.918</td>
<td>9.422</td>
<td>1.367</td>
<td>47</td>
</tr>
<tr>
<td>Total</td>
<td>690.148</td>
<td>65.099</td>
<td>8.509</td>
<td>197.716</td>
<td>9.876</td>
<td>1.618</td>
<td>40</td>
</tr>
<tr>
<td># tests</td>
<td>217M</td>
<td>217M</td>
<td>11M</td>
<td>217M</td>
<td>217M</td>
<td>217M</td>
<td>-</td>
</tr>
</tbody>
</table>

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.
## BrCounty and BrSoil

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre.</td>
<td>1.242</td>
<td>0.225</td>
<td>0.478</td>
<td>0.549</td>
<td>0.324</td>
<td>0.099</td>
<td></td>
</tr>
<tr>
<td>Inter.</td>
<td>1.444</td>
<td>0.152</td>
<td>0.015</td>
<td>0.385</td>
<td>0.040</td>
<td>0.018</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2.686</td>
<td>0.377</td>
<td>0.493</td>
<td>0.934</td>
<td>0.364</td>
<td>0.117</td>
<td></td>
</tr>
<tr>
<td># tests</td>
<td>300K</td>
<td>300K</td>
<td>70K</td>
<td>300K</td>
<td>300K</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## UsCounty and UsAquifers

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre.</td>
<td>7.884</td>
<td>0.812</td>
<td>2.628</td>
<td>1.610</td>
<td>0.392</td>
<td>0.164</td>
<td></td>
</tr>
<tr>
<td>Inter.</td>
<td>42.816</td>
<td>4.059</td>
<td>0.023</td>
<td>11.198</td>
<td>0.612</td>
<td>0.096</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>50.700</td>
<td>4.871</td>
<td>2.651</td>
<td>12.808</td>
<td>1.004</td>
<td>0.260</td>
<td></td>
</tr>
<tr>
<td># tests</td>
<td>13M</td>
<td>13M</td>
<td>159K</td>
<td>13M</td>
<td>13M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## UsCounty and UsCountyRotated

<table>
<thead>
<tr>
<th></th>
<th>Rational*</th>
<th>Interval*</th>
<th>CGAL*</th>
<th>Rational</th>
<th>Interval</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre.</td>
<td>14.532</td>
<td>1.422</td>
<td>7.482</td>
<td>2.798</td>
<td>0.454</td>
<td>0.251</td>
<td></td>
</tr>
<tr>
<td>Inter.</td>
<td>675.616</td>
<td>63.677</td>
<td>1.027</td>
<td>194.918</td>
<td>9.422</td>
<td>1.367</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>690.148</td>
<td>65.099</td>
<td>8.509</td>
<td>197.716</td>
<td>9.876</td>
<td>1.618</td>
<td></td>
</tr>
<tr>
<td># tests</td>
<td>217M</td>
<td>217M</td>
<td>11M</td>
<td>217M</td>
<td>217M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Thread 1

- \( o(v_1, v_2, v_3) \) \( \rightarrow -1 \)
- \( o(v_1, v_2, v_4) \) \( \rightarrow -1 \)
- \( o(v_3, v_4, v_1) \) \( \rightarrow -1 \)
- \( o(v_3, v_4, v_2) \) \( \rightarrow 1 \)

**intersections[thread_id] = 1;**

### Thread 2

- \( o(u_1, u_2, u_3) \) \( \rightarrow 1 \)
- \( o(u_1, u_2, u_4) \) \( \rightarrow 1 \)
- \( o(u_3, u_4, u_1) \) \( \rightarrow -1 \)
- \( o(u_3, u_4, u_2) \) \( \rightarrow 1 \)

**intersections[thread_id] = 0;**

### Thread 3

- \( o(w_1, w_2, w_3) \) \( \rightarrow -1 \)
- \( o(w_1, w_2, w_4) \) \( \rightarrow 1 \)
- \( o(w_3, w_4, w_1) \) \( \rightarrow 2 \)
- \( o(w_3, w_4, w_2) \) \( \rightarrow 1 \)

/** must be reevaluated using exact arithmetic on CPU**

---

**Acknowledgement:**

- [CAPES](https://www.capes.gov.br/)
- [CNPq](https://www.cnpq.br/)

---

39