Employing GPUs to accelerate exact 3D geometric computation

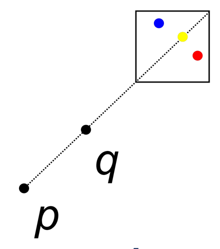
Salles V. G. Magalhães[1], W. Randolph Franklin[2], Marcelo Menezes[1]

[1]Universidade Federal de Viçosa, Brazil [2]Rensselaer Polytechnic Institute, USA

The challenge

- Roundoff errors: challenge in geometric computation.
- They can be avoided with exact rational numbers.
- Big datasets:
  - Greater chance of having errors.
  - Computation with rationals: slower than native floats.
- People want exactness and performance.

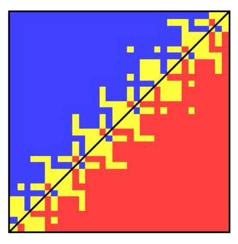
Example



1

$$\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix}$$

2



3

1. 2D orientation
2. Can be computed with a determinant
3. Errors due to floating-point arithmetic. Source of the image: [2]

Interval Arithmetic (IA)

- Interval arithmetic (IA) + arithmetic filtering can accelerate exact computation.
- Each coordinate/value: represented with exact part (rationals) and an interval approximation (floats).
- Computation is done with the approximation.
  - E.g.: [3,5] - [1,2] = [3-2,5-1] = [1.0,4.0]
    - the approximation [0.9,4.1] is **ok** (**contains** [1,4])
    - the approximation [1.1,4.1] is **not ok** (does not contain [1,4])
- Interval arithmetic + IEEE-754 (rounding modes): computation can be done ensuring the interval will always CONTAIN be exact result (containment property).
- Containment property → sign of the exact result can often be inferred from the intervals:
  - Is a*b - c = [1.0,4.0] positive? **Certainly** → use this result.
  - Is a*b - c = [-0.1,4.0] positive? **Maybe** → recompute with better approximations (double, rationals, etc).
- Geometric predicates: typically computed with sign of a determinant (suitable for IA).

IA on GPUs

- IA: much faster than rationals, but slower than regular floating-point.
- GPUs: excellent for **floating-point** and intervals.
- Rounding mode can be quickly switched (on a CPU → this would empty the pipeline).
- Example of the operator + using CUDA:
  - $[a_{lb},a_{ub}] + [b_{lb},b_{ub}] = [\underline{a_{lb} + b_{lb}}, \overline{a_{ub} + b_{ub}}]$

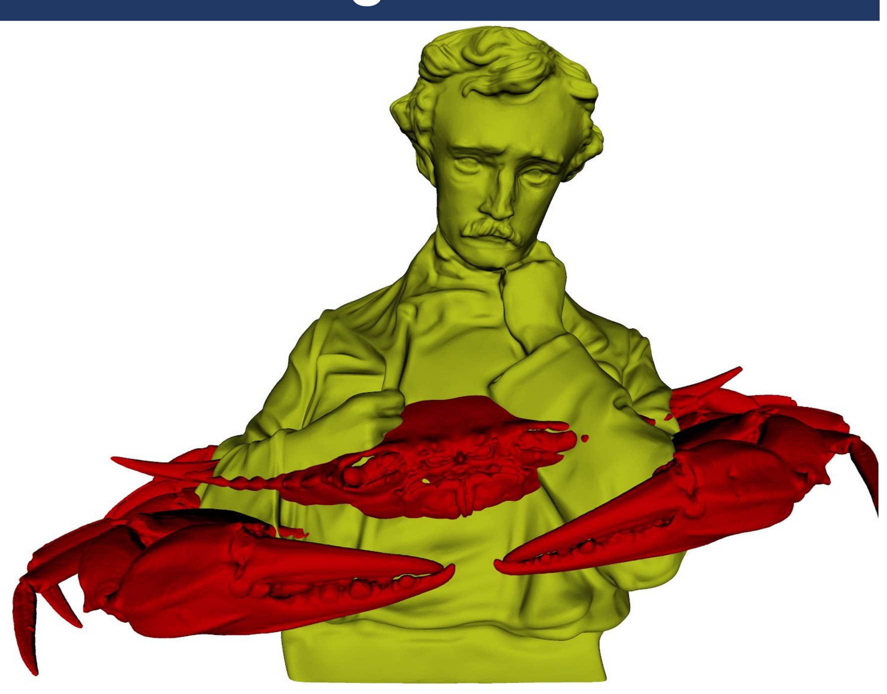
rounding up to the next representable **float**

```
#define INTERVAL_FAILURE 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_FAILURE;
    }

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

rounding up to the next representable **float**

Intersecting red and blue triangles

- Problem: find triangles from one mesh intersecting triangles from another one.
- Applications: collision detection, boolean operations, etc.
- Goal: compute it exactly and efficiently.
- Uniform grid index employed for avoiding testing $O(N^2)$ pairs of triangles.
- IA + rationals for exactness.
- GPU is employed for performance.

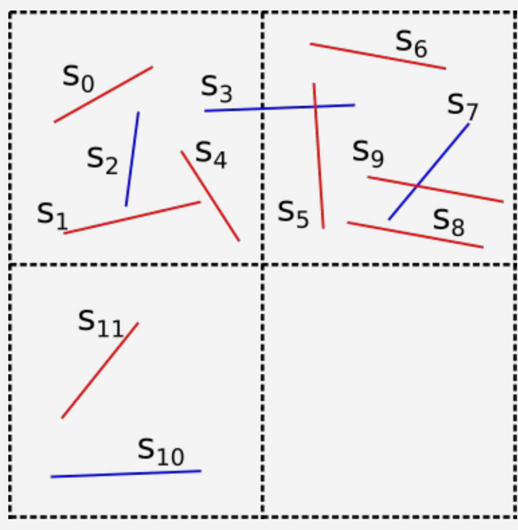


Two overlaid meshes: Blue crab and Edgar Allan (provided by IMR 2024)

Steps of the algorithm

1 - Uniform grid indexing

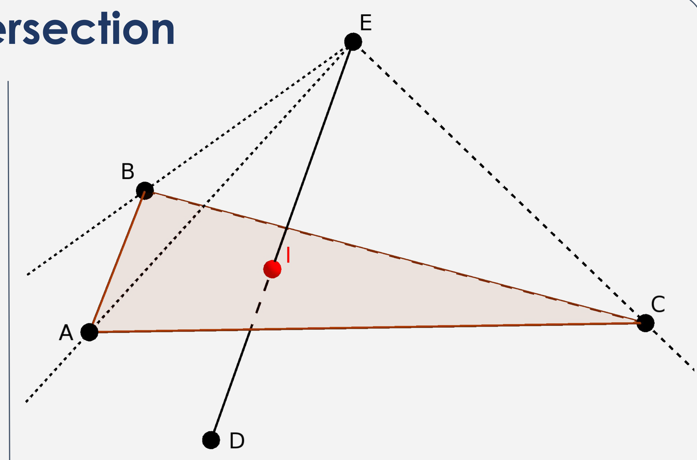
- 3D grid is created with a ragged array.
- Red and blue triangles inserted into the cells they intersect.
- For each cell c: bounding-box intersection tests are performed with the pairs of red-blue triangles in c.
- Bounding-box tests performed using two passes:
  - First: count the intersections.
  - Second: insert the intersecting pairs into an array.
- Each GPU thread processes some pairs.
  - Challenge: determine the pair each GPU thread will process (irregular distribution of triangles among grid cells).
- **Result**: array with pairs of potentially intersecting triangles.



2D example of a 2x2 uniform grid indexing red and blue segments.

2 - Triangle-triangle intersection

- For each pair of potentially intersecting triangles, intersection tests are performed.
- Uses orientation predicates implemented with IA.
- Orientation = sign of determinant: IA returns positive, negative, 0 or unknown (failure).
- Each GPU thread processes a pair of potentially intersecting triangles.
- Result is two arrays:
  - Intersections: certainly intersecting pairs of triangles.
  - Failures: Interval failures (rarely happens) - when orientation cannot be inferred using the intervals.



Intersection of a segment and a triangle can be computed with 5 3D orientations.

Intersection of a segment and a triangle → intersection of two triangles.

3 - Post-processing

- The (typically few) failures (uncertainties) are re-evaluated on the **CPU** with GMP rationals.
- Duplicated pairs of intersecting triangles are removed (using a **GPU** sort+unique implementation).

Results and conclusions

- Intel Xeon E5-2660 CPU at 2 GHz (3.2 GHz Turbo Boost), 256 GB of RAM, RTX 8000 GPU (48GB of RAM + 4608 CUDA cores).
- Datasets provided by IMR2024 and tetrahedralized with Gmsh:
  - Blue Crab: 25×10^6 triangles → 45×10^6 triangles in the ragged array
  - Edgar Allan (poet): 33×10^6 triangles → 64×10^6 triangles in the ragged array
- Uniform grid: 100^3 cells, 87% are empty
- Baseline: sequential CPU implementation
- Steps:
  - **Pre-processing**: access index, perform bounding-box tests and distribute work among threads (GPU version)
  - **Intersection**: perform intersection tests with orientation predicates
  - **Post-processing**: remove duplicates and re-evaluate interval failures with rationals

Dataset	*BlueCrab* vs *EdgarPoet*		
Method:	CPU	GPUDouble	GPUFloat
	Time (s)		
Pre-processing	64.86	1.09	1.09
Intersection	325.52	11.80	0.33
Post-processing	8.08	0.11	0.63
Data transfer	-	1.75	1.97
Total time (s)	398.46	14.75	4.02
#interval failures	-	0	267,238
#bounding-box tests	$14,754.9 \times 10^6$		
#intersection tests	771.5×10^6		
#intersections	89.5×10^6		

- Speedup: 993x on the intersection tests, 99x on the total time.
- Double precision: fewer (0) filter failures, but slower computation.
- Approximate floats on GPUs (where they shine) can accelerate exact geometric computation.
- Future work:
  - Employ this technique for other applications.
  - Higher speedups could be achieved in applications where bigger bottlenecks could be moved into the GPU (performing more computation and fewer memory transference)

Bibliography

1. Marcelo Menezes, Salles Magalhães, Matheus Oliveira, W. Randolph Franklin, Rodrigo Chichorro. Fast Parallel Evaluation of Exact Geometric Predicates on GPUs. Computer-Aided Design 2022; 150
2. Kettner Lutz, Mehlhorn Kurt, Pion Sylvain, Schirra Stefan, Yap Chee Keng. Classroom examples of robustness problems in geometric computations. Comput Geom 2008;40(1):61–78

Acknowledgement

CAPES

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

W. Randolph Franklin:

Salles V. G. Magalhães: