11-2020

Efficient Filters for Geometric Intersection Computations using GPU

Yiming Liu
Marquette University

Satish Puri
Marquette University, satish.puri@marquette.edu

SIGSPATIAL '20: Proceedings of the 28th International Conference on Advances in Geographic Information Systems, (November 2020): 487-496. DOI. This article is © Association for Computing Machinery (ACM) and permission has been granted for this version to appear in e-Publications@Marquette. Association for Computing Machinery (ACM) does not grant permission for this article to be further copied/distributed or hosted elsewhere without the express permission from Association for Computing Machinery (ACM).

Efficient Filters for Geometric Intersection Computations using GPU

Yiming Liu
Marquette University, USA
Satish Puri
Marquette University, USA

Geometric intersection algorithms are fundamental in spatial analysis in Geographic Information System (GIS). Applying high performance computing to perform geometric intersection on huge amount of spatial data to get real-time results is necessary. Given two input geometries (polygon or polyline) of a candidate pair, we introduce a new two-step geospatial filter that first creates sketches of the geometries and uses it to detect workload and then refines the sketches by the common areas of sketches to decrease the overall computations in the refine phase. We call this filter PolySketch-based CMBR (PSCMBR) filter. We show the application of this filter in speeding-up line segment intersections (LSI) reporting task that is a basic computation in a variety of geospatial applications like polygon overlay and spatial join.

We also developed a parallel PolySketch-based PNP filter to perform PNP tests on GPU which reduces computational workload in PNP tests. Finally, we integrated these new filters to the hierarchical filter and refinement system to solve geometric intersection problem. We have implemented the new filter and refine system on GPU using CUDA. The new filters introduced in this paper reduce more computational workload when compared to existing filters. The processing rate of the new filter and refine system for line segment intersection reporting task is 61 million/sec on average.

Keywords

Theory of computation→Computational geometry, Information systems→Geographic information systems, Computing methodologies→Massively parallel algorithms, HPC, Parallel Algorithms, CUDA, Spatial Operations

1 Introduction

Filter and refine strategy is used in many spatial computing algorithms for spatial query, spatial join and overlay in geographic information system [10]. Given two input layers of geometries, filter step uses minimum bounding rectangle (MBR) approximation of a geometry and refine step uses the actual vertices that represent it. Typically, filter step is lightweight and refinement step is compute-intensive because of complex computational geometry algorithms. Filter step produces candidate pairs which may result in false hits. The refinement step further examines the candidates sequentially to eliminate false hits by using computational geometry algorithms on each candidate. Geometric intersection algorithms are fundamental in spatial computing [7, 9, 13, 17, 22]. Here we study GPU-based implementation of filter and refine strategy which is relevant to spatial join and polygon overlay algorithms.

Geometric intersection algorithms for polygons use line segment intersection (LSI) and point-inpolygon (PNP) operations as building blocks. Here, we are interested in the LSI reporting problem which means that all the points of intersection between two polygons should be reported. Moreover, for all the points of two polygons corresponding to a candidate, the inside/outside status needs to be determined. LSI and PNP operations are useful for implementing boolean set operations like union, intersection, and difference for a pair of polygons. Polygon intersection and polygon clipping algorithms internally invoke LSI and PNP tests [7, 9, 24]. Depending on the variation in size of line segments and its spatial distribution, planesweep [14] and grid-based algorithms on CPU [12] and GPU [21, 23] have been reported in literature. In addition, there are data structures like segment tree and R-tree that have been used to speedup LSI problems [19].

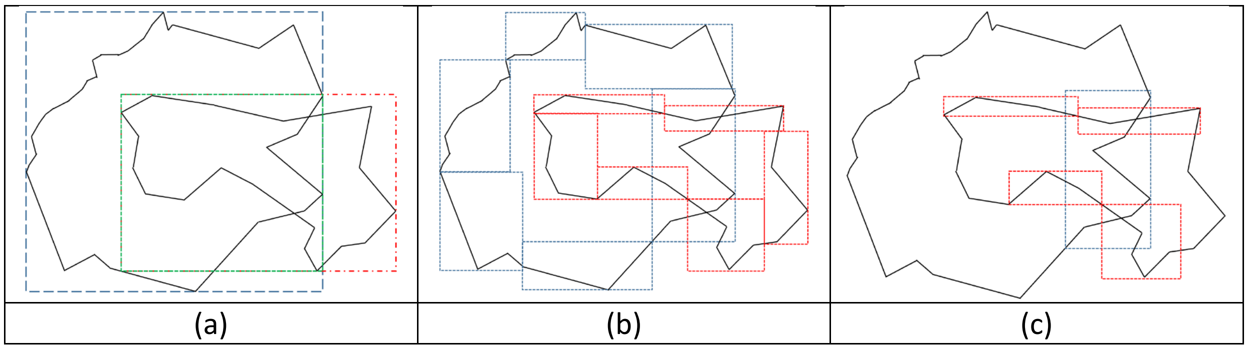

Fig. 1. Two input polygons with (a) CMBR (green rectangle), (b) PolySketch showing the tiles and (c) only overlapping tiles after applying PolySketch filter.

In addition to MBR, other approximations which are used as filter are rotated minimum bounding box, convex hull, minimum bounding circle and ellipse, n-cornered bounding polygon, etc [5, 6, 20]. Recent approaches for implementing spatial computations using GPU, include a variety of filters, for instance, Common MBR Filter (CMF) [2, 4], two-level uniform grid [12], Grid-CMF [1], and PolySketch [11]. A collection of filters applied in a hierarchical manner for speeding up geometric intersection on GPUs was presented as Hierarchical Filter and Refine (HiFiRe) technique [11]. In this paper, we extend the filter and refine technique by adding two efficient filters for speeding up LSI and PNP operations. The filters are designed to exploit the parallel GPU architecture and minimize the workload inherent in the refinement step of spatial computations by improving the filter efficiency.

PolySketch filter uses sketch of a geometry represented by a set of contiguous MBRs (tiles) that approximate the geometry [11] instead of a single MBR. Common MBR filter is based on the common area of overlap between MBRs of the two input polygons of a candidate. The first new filter proposed here combines the strengths of PolySketch and CMBR filters and thus we refer to it by PolySketch-based CMBR (PSCMBR) filter. The second new filter is a PNP filter that uses PolySketch representation of the geometries to quickly find whether the points of a geometry are inside or outside of a given polygon. The contributions of this paper are as follows.

- PSCMBR Filter: Compared to standard R-tree filter, the PSCMBR filter discards on average 76% of candidate pairs which do not have line segment intersection points. The workload after using it is on average 98% and 90% smaller than using CMF and PolySketch filter respectively.
- PolySketch-based PNP filter: The workload after using it is on average 60% smaller than using Stripe-based PNP filter [11]. The workload after using tile-based PNP filter is on average 98% smaller than using constant vertex PNP filter [11].
- With the improved HiFiRe system equipped with new filters, we get on average 7.96X speedup compared to our prior version of HiFiRe system. The processing rate of this new filter and refine system for reporting line segment intersection is 61 million/sec on average for real datasets.

The rest of the paper is organized as follows. Section 2 describes the background and related work. Section 3 describes PSCMBR. Section 4 describes the PNP filter based on PolySketch. Section 5 shows the experimental results. Finally, we conclude the paper.

2 Background and Relatedwork

Filter and refine is a widely used technique that takes a two-step approach of first filtering the geometries that can potentially become part of the output using rectangular approximations (MBR). Given a collection of geometries in an input dataset (layer), each geometry is represented as one rectangle that encloses it completely in the filter phase. It has been shown that approximations excluding MBR are costly to construct [20]. After the filter phase, refinement is done using actual line segments of the input. This idea has been shown to be effective on a GPU which is a massively parallel hardware that accelerates the filter and refine computations. Using PolySketch as one of the filters, a hierarchical filter and refinement system (HiFiRe) [11] was implemented which is essentially a collection of filters to speedup geospatial intersection algorithms on a GPU. As shown in HiFiRe, the filter step is the key to improve the performance.

An algorithm for polygon intersection with N and M line segments require s$O(N.M)$ line segment intersection tests. This quadratic time complexity makes it an expensive operation. In GIS, millions of polygon intersections are common. As such, filters are designed which acts as proxy for polygons. One such filter is an MBR filter. Minimum bounding rectangle is the smallest axis-aligned rectangle that encloses a spatial object, such as a polygon. It only requires two multi-dimensional points to store a MBR. Computing a MBR and checking a MBR with other MBRs are much cheaper than checking the intersection points between line segments so MBR is widely used to the filters of basic spatial operations, such as the intersection test.

Common Minimum Bounding Rectangle (CMBR) is an approach based on MBRs and it is the overlapping area of MBRs of two polygons. The paper [2] introduces and applies it to perform spatial join for real datasets. As shown in Figure 1(a), the blue and red rectangles are MBRs that each encloses a polygon and the green rectangle is their CMBR. **Common MBR filter (CMF)** is an efficient filter based on the idea of CMBR for line segment intersection because it can ignore the line segments that do not overlap with the CMBR, which can reduce the computational workload in LSI refinement. In addition, it is possible that two polygons do not overlap but their MBRs overlap. CMF can identify this scenario and avoid expensive polygon intersection. This is the rationale of using a filter. However, the feature of CMBR makes it less effective if the CMBR is large as shown in Figure 1(a). Most of line segments are still in CMBR and cannot be ignored. In addition to CMF method, to improve the efficiency of MBR, the paper [18] introduces the clipped bounding box (CBB) that includes a set of clip points that clip away empty corners of MBRs. Danial et al. presented two spatial filters, namely, CMF and Grid-CMF. CMF is based on common MBR area between two cross-layer polygonal MBRs [1, 2]. Grid-CMF further partitions the Common MBR area. Both filters have been used in spatial join using GPU to filter out candidate pairs that do not need further refinement.

PolySketch is a representation of a spatial object by a set of tiles [11]. A tile is a subset of consecutive vertices of a geometry and tile-size is the number of vertices in the tile. Once tile-size is fixed, then a MBR is calculated which is known as tile-MBR. If a geometry has n vertices and the tile size is set as b, it consists of n/b tiles. The basic idea is to first do tile intersections that are cheap and then to do expensive refinement on tiles that have overlap. Figure 1(b) shows an example of PolySketch. We can see that each polygon contains some tiles which contain different line segments. The performance of PolySketch is affected by the tile-size. Using different tile-sizes, we can get more or fewer tiles.

Generally, by using smaller tiles, we may discard more parts of polygons. We can discard the tiles whose MBRs do not overlap with others. Figure 1(c) shows the candidate tiles. The line segments within one tile should be compared with the line segments of other overlapping tiles.

CMF vs PolySketch: CMF is different from PolySketch filter because all line segments overlapping the CMBR of two polygons should be compared against each other in CMF. PolySketch can better handle the case where CMBR is large [11]. PolySketch filter checks every tile of a polygon with all tiles of another polygon. By using smaller tiles, the line segments within a tile are only compared with the line segments of the overlapping tiles. Most parts of polygons which cannot have intersection points can be safely ignored. However, the line segments within a tile cannot be discarded if a tile overlaps with others. All line segments within this tile should be tested. In short, there is room for further improvement in PolySketch filter when the number of candidate tile pairs is high and each tile contains a large number of line segments. In contrast to PolySketch, CMBR can contain any number of line segments. All line segments which do not overlap with the CMBR will be discarded. PolySketch has been compared to CMF in [11].

Another prior work in this area uses PixelBox technique which is pixel approximation of polygons for computation [21]. Geometries represented as 2D co-ordinates are converted to raster format (pixels) to leverage image processing using a GPU [8]. Another work by Audet et al. [3] uses uniform grid for polygon overlay. Space division techniques like gridding can potentially increase the problem size by replicating the line segments crossing the grid boundaries. In a filter and refine algorithm, planesweep technique is used in the refine phase when the dataset fits in the memory. Geometric intersection using GPU has been studied earlier for planesweep algorithm [15].

3 Polysketch-Based Common Mbr (Pscmbr) Filter

3.1 Overview of PSCMBR Filter

As we discussed before, CMF and PolySketch filter were used in the geometric intersection computations. These filters have their advantages and drawbacks. Our new PSCMBR filter is a more efficient filter that can handle various types of polygons with varying degree of overlaps. It combines the strength of CMF and PolySketch Filter. PSCMBR first creates sketches of the geometries. Then, it checks which tiles of a polygon overlap with tiles of another polygon and the overlapping tiles are candidate tile pairs. The tiles that do not overlap with other tiles are discarded. After this, we calculate the common area of overlap for every candidate tile pairs and check whether both tiles have line segments overlapping with the CMBR. Those candidate tile pairs that do not have any line segments in the CMBR are discarded. If they do, we will perform LSI function only for the line segments overlapping with the CMBR instead of all line segments within the tiles. So, in essence, CMBR filtering is used at the granularity of tiles instead of polygonal MBRs.

We illustrate PSCMBR filter in Figure 1. Figure 1(b) is the first step in using PSCMBR where we create sketches of polygons. There are four candidate tile pairs shown in Figure 1(c), where one tile of a polygon overlaps with four tiles of another polygon. Then, PSCMBR filter calculates the CMBRs of a pair of tiles corresponding to the four candidate pairs. The four rectangles in Figure 2 are their CMBRs. The two candidate tile pairs whose CMBRs are yellow do not need further refinement because only one tile has line segments overlapping the CMBR. Since another polygon does not have any line segment overlapping the CMBR, there cannot be any line segment intersection. Therefore, we do not need to perform refinement phase. Another two candidate tile pairs whose CMBRs are purple need further refinement because both polygons have line segments overlapping their CMBRs. In addition, only line segments overlapping the purple rectangles need to be checked instead of all line segments within the tiles. This leads to reduction in workload in the refinement phase.

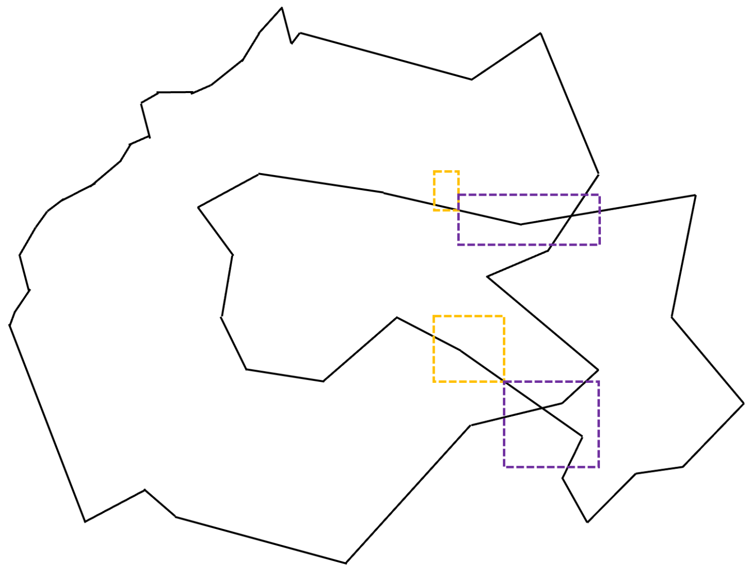

Fig. 2. PSCMBR filter with four tile-CMBRs. Only the common area of overlap between the candidate tile pairs are retained (see Figure 1(c)).

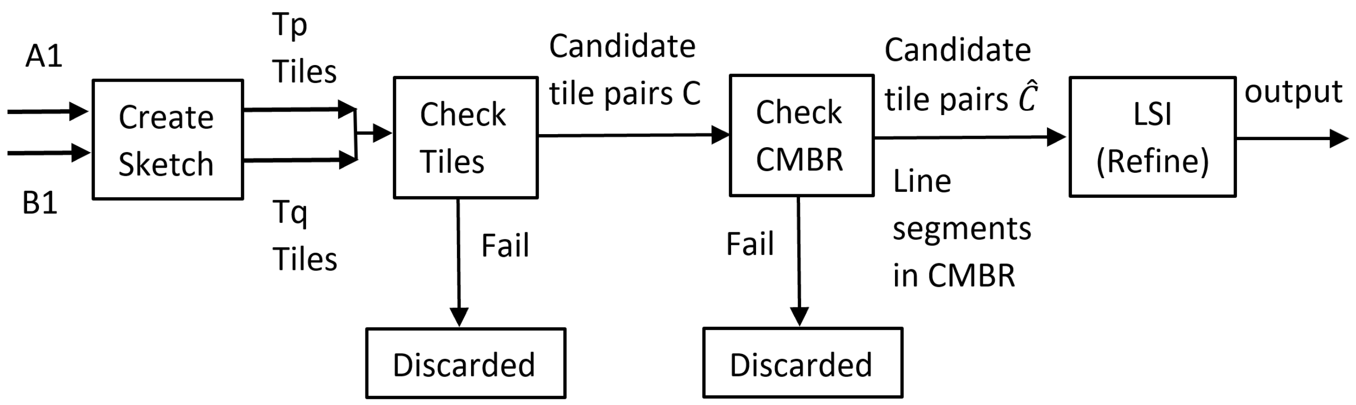

Fig. 3. PSCMBR filter for reporting line segment intersections (LSI). Two polygons A1 and B1 are the input and the output is list of intersections.

Execution time model of intersection of two geometries using PSCMBR:We describe the workload first in terms of tile-MBR intersections (two filters) and refinement using LSI. Table 1 defines the symbols used in the time complexity of intersection of two geometries (polygons). T_P and T_Q are tile-counts for two polygons with P and Q numbers of line segments respectively. $\frac{P}{T_P}$ and $\frac{Q}{T_Q}$ are number of line segments in a tile (tile-size) of the respective geometries.

Figure 3 shows the data-flow and control-flow in PSCMBR filter and refine system. Input of PSCMBR is the candidate tasks. As shown in Figure 3, A1 and B1 are two polygons of a task. Output of the 'check tiles' step is a collection of candidate tile-pairs of size C. Output of the 'check CMBR' step is a collection of candidate tile-pairs of size $\hat{C}$. Because of CMBR filtering on candidate tile pairs, we have $\hat{C} <= C$. Moreover, $\hat{P} <= P$ and $\hat{Q} <= Q$. Using PolySketch, the time complexity is given by the following equation:

$$\mathcal{T} = T_P.T_Q.\mathcal{T}_{MBR} + C.\frac{P}{T_P}.\frac{Q}{T_Q}.\mathcal{T}_{LSI} \quad (1)$$

In Figure 4, we show the relationship between tile-size and effectiveness of the PolySketch filter. Maximum number of tiles in a polygon is bounded by the number of line segments in the polygon

Table 1. Symbol Table

$Symbol$	$Definition$
$\mathcal{T}_{MBR}$	Time for checking if two MBRs overlap
$\mathcal{T}_{CMBR}$	Time for checking if a line segment overlaps CMBR
$\mathcal{T}_{LSI}$	Time to find intersection point of two line segments
P, Q	Number of line segments in two input geometries
T_P , T_Q	Number of tiles in the two input geometries
C	Number of candidate tile pairs after PolySketch
$\hat{C}$	Number of candidate tile pairs after CMBR
$\hat{P}, \hat{Q}$	Number of line segments in CMBR

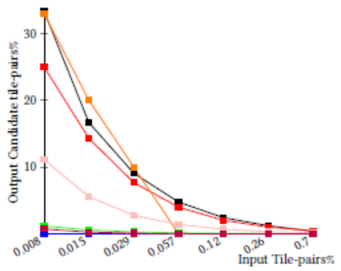

Fig. 4. Effect of tile-size on the number of candidate tile-pairs. Input Tile-pairs% $= \frac{T_{p*}T_q}{P*Q} * 100$. Candidate tile-pairs% $= \frac{C}{T_{p*}T_q} * 100$. Each line denotes the output candidate tile-pairs% for a given input polygon-pair.

because we do not split a line segment. Therefore, for calculating the input tile percentage, we use the product of P and Q. Tile-size determines the number of tiles used in the filter. Large (small) tile-sizes correspond to very small (large) number of input tiles. In general, small tile sizes lead to better filter efficiency. However, small tile size means a larger number of tiles which increases the overhead of the filter. So, there is a tradeoff between filter efficiency and time complexity of the filter. The polygon pairs used here have been taken from Classic[1], Ocean[2] and Water (described in Subsection 5.1). Table 2 shows an example of input polygon pair of Figure 4. For spatial join and polygon overlay workloads, it is difficult to estimate good tile size because as we can see the output-size varies from one candidate pair to another.

Finding whether two rectangles overlap requires a single line of code. However, finding the point of intersection of two line segments is more than 10 lines of code. As such, $\mathcal{T}_{MBR} \ll \mathcal{T}_{LSI}$. In addition, $\mathcal{T}_{CMBR} \ll \mathcal{T}_{LSI}$. From Equation 1, the LSI workload depends on the candidate tile-pairs.

Table 2. An example of input polygon pair in Figure 4

P	Q	Tile-size for A1	Tile-size for B1	Input tile-pairs %
72997	101242	128	128	0.006%
72997	101242	15	15	0.44%

Overall performance depends on the delicate balance between the tile-MBR intersections and line segment intersections as shown in Equation 1.

Using PSCMBR, the time complexity is given by the following equation:

$$\mathcal{T} = T_P . T_Q . \mathcal{T}_{MBR} + C . \frac{P}{T_P} . \frac{Q}{T_Q} . \mathcal{T}_{CMBR} + \hat{C} . \hat{P} . \hat{Q} . \mathcal{T}_{LSI} \quad (2)$$

3.2 Advantages of PSCMBR Filter

As we discussed before, the existing CMF is not efficient if the CMBR of two polygons is very large. In contrast to CMF, PolySketch can handle this case well by using small tiles. In addition, using CMF also requires calculation and storage of the line segments overlapping the CMBR, which also takes more time if the CMBR is large. However, if the CMBR of two polygons is small, CMF works better than PolySketch because the tiles of PolySketch contain a fixed number of line segments and we can only ignore or keep all line segments within the same tile. CMBR can contain any number of line segments.

PSCMBR can solve the previous problems. Since it creates the sketches of polygons, it can discard most parts of polygons, which do not overlap with another polygon. Then, checking whether both tiles have line segments overlapping with their CMBR can reduce the number of candidate tile pairs. The new filter can potentially reduce the number of false hits compared to the classical filter and refine strategy, CMF and PolySketch filter. Moreover, if both tiles still have line segments overlapping with their CMBR, we only need to perform the refinement phase for the line segments inside the CMBR, instead of all line segments in a tile. Therefore, the refinement phase has to handle fewer line segments.

For CMF, storing all line segments overlapping with the CMBR of two polygons for all tasks has significant memory overhead, especially on GPU. There are variations in vertex count and degree of overlap in real-world datasets. For example, some polygons are huge (about 50,000 vertices) and some polygons are small (about 50 vertices). The CMBR of two huge polygons can be also huge. The global memory in a GPU is limited. PSCMBR can handle this scenario because it calculates the CMBR of only candidate tiles which leads to better space complexity. In addition, different GPU threads can be assigned to process different tiles of the same polygon, which increases the parallelization.

3.3 The Implementation of PSCMBR Filter

In CUDA programming model, the threads are organized as blocks of threads. A thread block may include up to 1024 threads. $threadIdx$ variable stores a unique thread ID assigned by CUDA to each GPU thread. It is the index of the current thread within its block. $blockIdx$ variable is a unique block ID assigned by CUDA to each GPU thread block. A CUDA kernel is a function that runs on a GPU. It is executed in parallel by different threads. Threads in the same block can communicate by shared memory or global memory.

Figure 5 shows the pseudo-code of PSCMBR filter applied to LSI function using CUDA. Each candidate polygon pair (task) will be assigned a thread block. Here we are showing how the filter is implemented for a single task that uses a single GPU thread block for simplicity. Our actual kernel applies the same filter on thousands of such tasks by mapping one thread block to one candidate pair. For parallelization, we define a task as a pair of polygons whose MBRs overlap.

In the CUDA pseudo-code, the number of tiles for the two polygons are stored in arrays $numTileL1$ and $numTileL2$, and the number of line segments in a tile are stored in $tileSize1$ and $tileSize2$. Given the line segments of the two input polygons denoted by arrays $segment1$ and $segment2$, a tile needs two offsets to mark the starting and ending points in the arrays. We can find the corresponding line segments within the tiles by using them. These offsets are stored in the arrays $prefixSum1$ and $prefixSum2$. The MBRs of tiles are stored in $tileMBR1$ and $tileMBR2$ arrays.

```
#define SIZEl 30
#define SIZE2 30
__global__ void PSCMBR( int numTileLl,int numTileL2,
     int tileSizel , int tileSize2 , int *prefixSuml,
     int *prefixSum2 , MBR *tileMBRl , MBR *tileMBR2 ,
     LineSegment *segmentl, LineSegment *segment2, ... ){
   //We consider the larger polygon as the 1stpolygon
   //numTilell is the number of tiles of the 1st polygon
   for (int j=threadidx.x; j<numTileLl; j+=blockDim.x){
     for (int k=0; k<numTileL2; k++)
       if(tileMBRl[j] overlaps tileMBR2[k]){
         LineSegment subSegmentl[SIZE1],subSegment2[SIZE2];
         int count1=0;
         int count2=0;
         //calculate the CMBR of two tiles
         MBR cmbr=getCMBR (tileMBRl [ j ], tileMBR2 [ k]);
         for (int jj=0; jj<tileSizel; jj++){
           if (segmentl [ prefixSuml [ j ]+jj ] overlaps cmbr ){
              storeSegment (subSegmentl [ countl],
                                  segmentl [ prefixSuml [ j ]+jj ]);
              countl++;
           }
          }
          if ( count1 !=0){
            for (int kk=0; kk<tileSize2; kk++){
              if (segment2 [ prefixSum2 [ k]+kk ] overlaps cmbr ){
                 storeSegment (subSegment2 [ count2 ],
                                      segment2 [ prefixSum2 [ k]+kk ]);
                 count2++;
                 }
          }
          if ( count2 !=0){
            //Calculate and store intersection points
            LSI (subSegmentl, subSegment2, countl, count2 );
          }
        }
      }
    }
  }
}
```

Fig. 5. CUDA Implementation of PSCMBR Filter

For simplicity, given a candidate pair, let us assume that the 1st polygon in the algorithm is the one that has more tiles as described in Subsection 3.4. When we use CUDA, the first step is to create different thread blocks for different tasks. Then, we assign the threads to the polygon which has more tiles and every tile will be compared with all tiles of another polygon. If two tile-MBRs overlap, we will calculate their CMBR and check if both tiles have line segments overlapping with the CMBR. The implementation follows the algorithm that we discussed earlier. SIZE1 and SIZE2 in Figure 5 are always larger than the tile-size to prevent buffer overflow.

In the CUDA algorithm, it is possible to increase the number of threads to avoid the k loop which is sequential. However, the kernel handles a large number of tasks with different polygon sizes and we assign one thread block to one task. In the real-world datasets, candidate polygon-pairs (tasks) have different vertex count (e.g., 50K or 100), so it is difficult to decide how many threads to be used in each block for a huge number of tasks. While mapping tiles to threads, we make sure that the inner k loop goes over fewer number of tiles from the smaller polygon.

3.4 Optimization: Mapping tiles to threads

In the implementation, given a candidate pair, a thread picks a tile of a polygon and compares it with all tiles of another polygon. When mapping computations associated with tiles to GPU threads, we assign threads to the tiles of the larger polygon. Our implementation dynamically swaps the order of polygons in a candidate pair based on the number of tiles in a polygon. This leads to a better division of work among threads and it leads to better mapping of the computations to different levels of GPU parallelism. For example, to illustrate this optimization, let us assume that we have a thread block with 128 threads and we have two polygons A with 256 tiles and B with 16 tiles. If we assign 128 threads to A, a tile of A should be compared with only 16 tiles of B by each thread. However, if we assign 128 threads to B, we cannot make full use of the threads and a tile of B should be compared with 256 tiles of A by each thread. This increases the workload for all threads. Therefore, we compare the number of tiles of two polygons from layer 1 and layer 2. Then, we assign threads to the polygon which has more tiles. This helps us in making better use of the GPU resources and implement the algorithm efficiently. In addition, the workload in every thread is more balanced.

4 Point-In-Polygon Filter Using Polysketch

Point-in-Polygon (PNP) tests are necessary for polygon-polygon intersection algorithms to create the output polygon. For instance, when a polygon is completely inside another polygon, there are no intersection points. A brute-force algorithm requires running PNP test for every point, which is an expensive operation. Therefore, we have designed a PNP filter based on PolySketch approximation. We assume that PNP filter is invoked after LSI function as discussed earlier.

For a candidate polygon pair (A,B), the input to the algorithm are 1) two list of vertices from each polygon and 2) information about intersecting points found by the LSI function. The goal is to find which points of a polygon A fall inside/outside of polygon B and vice versa. The classical point-in-polygon (PNP) test for a point is $O(N)$ where N is the number of points in a polygon. The basic idea is to create a filter that minimizes the number of expensive PNP tests using PolySketch.

Basic Idea: Using Jordan curve theorem, we can show that the inside/outside status of points of a polygon A changes when its line segments intersect with the line segments of polygon B. This idea is utilized in polygon clipping algorithms [9, 16] to avoid expensive PNP tests by first inserting the segment intersections into the original polygons to create a graph and then traversing the graph to find the inside/outside status of the points of input polygons. When polygons are represented as tiles (a subset of contiguous vertices), this idea leads to a new PNP filter.

4.1 Algorithm Overview

There are three cases that need to be handled for a candidate pair.

Case I: If a tile's MBR overlaps with another tile's MBR and there are line segment intersections, the vertices inside these two tiles need further processing. This is the case where the filter does not help. So, PNP tests are required for all the vertices in the tiles.

Case II: If a tile's MBR overlaps with another tile's MBR but there is no line segment intersection, the inside/outside status of the vertices in a tile should be the same.

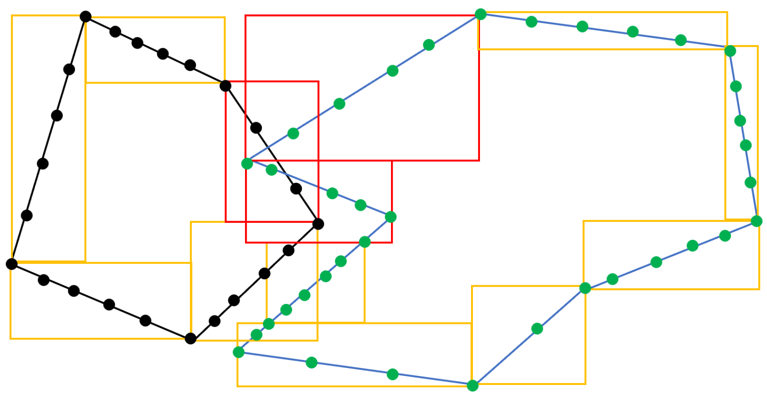

Fig. 6. An example of intersection tile (red rectangle) and no-intersection tile (yellow rectangle)

Case III: If a tile's MBR does not overlap with any other tiles, then the inside/outside status of the vertices in a tile should also be the same.

In the second and the third cases, the filter works in reducing the number of PNP tests because only one PNP test is required for an entire tile. The status of the remaining vertices of a tile is the same. Therefore, PolySketch-based PNP function divides the tiles into two types: $intersection\ tile$ and $no-intersection\ tile$. If a tile does not have any line segment intersection, we consider this tile as the $no-intersection\ tile$. If a tile has at least one line segment intersection, we consider this tile as the $intersection\ tile$. For $no-intersection\ tile$, we need a single PNP test and then we know whether all vertices within this tile are inside another polygon or not. For $intersection\ tile$, we test all vertices within a tile because there are line segment intersections; so the status of the points before the intersection-point will be different from the status of the points that are after it if we traverse the points in a clockwise order. We have observed that the number of intersection-points are far less than the input size of the overlapping polygons. Therefore, this limitation has very less impact on the performance.

As shown in Figure 6, there are two polygons C1 (black) from layer1 and D1 (blue) from layer 2. The tile-size is set as 5 line segments. We can see there are five tile-MBR overlap pairs. For two tile-MBR overlap pairs, they have line segment intersections. For the other three tile-MBR overlap pairs, they do not have any line segment intersection. Therefore, we should do the PNP test for all vertices of only one tile of C1 and two tiles of D1. For other tiles, we run the PNP test for only a few vertices within every tile.

4.2 Comparison of PNP workload reduction

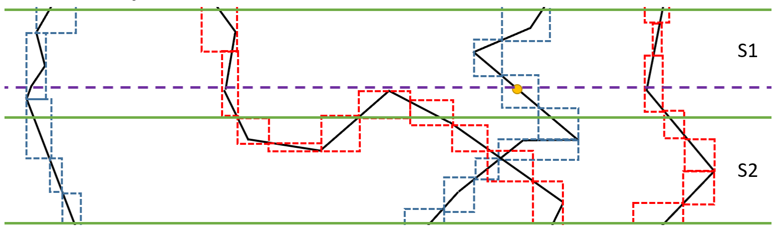

Fig. 7. Illustration of PNP functions. Two polygonal chains extracted from input polygons is highlighted by red and blue colored tiles. The space is divided into two stripes S1 and S2. Dotted line is an imaginary ray parallel to X-axis and passing through the test point shown as yellow point.

Comparsion with our prior work: In our previous work [11], we had used space division (striping) to decrease the PNPworkload wherewe divided the space occupied by the two overlapping polygons into horizontal stripes and mapped the line segments of the polygons to the stripes based on overlap. This mapping step used extra memory to store the line segments contained in the stripes and was done on a multi-core CPU as a pre-processing step. In addition, if the area to be divided is very large or the number of stripes is high, then the performance degrades. For real datasets with a variety of candidate pairs, using a static number of stripes limited the performance. Another idea that we utilized was to do only a few PNP tests when it was determined that a polygon is contained completely inside another polygon. Overall, the PNP part was the bottleneck in earlier work [3, 11]. The for-loop in the rayshooting algorithm was parallelized in [2] to improve the performance. Therefore, we revisited the parallel PNP algorithm in this paper. We do not use CPU-based preprocessing in this paper using a novel approach. Our new algorithm reduces the workload considerably and performs better than [11].

Stripe-based PNP function: Stripe-based PNP function was proposed in the paper [11]. Using ray casting algorithm for the PNP test, a point is tested whether it is inside another polygon based on how many times an imaginary ray from a test point crosses the polygon boundary. For a test point, we can reduce the PNP test workload by discarding those line segments which the ray could not cross. This is implemented by comparing y-coordinates of line segments with the test point. Figure 7 shows how Stripe-based PNP function works. In this example, the area is divided into two stripes S1 and S2 (area between two green lines). In short, we divide the area considering the y-coordinates of the polygonal vertices. For both polygons, we check every vertex whether it is inside any stripe and every line segment whether it crosses the stripe boundaries. Then, the vertex inside one stripe is compared to another polygon's line segments corresponding to the same stripe.

Tile-based PNP function: In our new approach, we compare the test point only with the line segments within the tiles whose MBRs overlap with the y-coordinate of the test vertex. If a tile's MBR does not overlap with it, we can discard the line segments within the tile for this vertex. If a tile's MBR overlaps with it, we compare the vertex with all line segments within this tile. The situation is different for different tiles and different tasks according to the tile size and the number of vertices of polygons. This technique can reduce a lot of line segments even if the polygon is large or huge. For small polygons, it can also reduce a similar number of line segments compared with using striping. Figure 7 shows a part of two polygons. The areas between green lines are different stripes and there are two stripes. If we use striping, the test vertex (yellow) should be compared with all line segments of another polygon in S1. By using tile-based PNP function, it should be compared with the line segments within only two tiles (considering y coordinate of the test vertex).

4.3 The implementation of PNP filters

For our system, we have two different kernels to perform PNP tests. One kernel (K1) is for the tasks where one polygon may be completely inside another polygon since two polygons do not have any line segment intersection. Another kernel (K2) is for the tasks where two polygons have intersection points. For K1, tile-based PNP function is used since two polygons do not have any line segment intersection. PolySketch-based PNP function cannot be used here. For K2, PolySketch-based PNP function is used for tasks where two polygons have line segment intersection points. In addition, tile-based PNP function can be also used here to reduce the workload. Therefore, we apply these two filters together in the same kernel.

5 Experimental Result

5.1 Data sets

We have used three datasets to evaluate our system: (1) Water, (2) Urban, and (3) Lakes. The details are shown in Table 3. Urban and Water are from http://www.naturalearthdata.com and http://resources.arcgis.com. The third dataset (Lakes) is from http://spatialhadoop.cs.umn.edu/datasets.html.

Table 3. Three real datasets used in our experiments

Label	Dataset	Polygons	Size
Water	USA_Water_Bodies	463,591	520MB
	USA_Block_Boundaries	219,831	1300MB
Urban	ne_10m_urban_areas	11,878	20MB
	ne_10m_states_provinces	4,647	50MB
Lakes	Lakes	7.5M	9GB
	Sports	1.8M	590MB

5.2 Hardware Description

We have used a single Nvidia Titan V GPU to run the experiments. Titan V has Volta architecture. It has 12 GB HBM2 memory, 5120 CUDA cores and its memory bandwidth is 652.8 GB/s. We have also used Intel Xeon E5-2695 multi-core CPU with 45MB cache and base frequency of 2.10GHz.

5.3 PSCMBR Filter Performance Results

Given two layers of polygons, the input to all the filters is the set of candidate polygon pairs obtained from R-tree query using standard MBR filter. To compare the performance of filters, let us consider that we have t tasks and each task has two cross-layer polygons. A task refers to a candidate polygon pair. In this paper, we compare the new filter PSCMBR with PolySketch (PS) filter and Common MBR filter (CMF).

First, we show the results of the workload in the refinement phase after the application of a filter. We calculate the refinement workload for each task first and then add the workload for all tasks to get total workload which is shown in the tables. For CMF, refinement workload for one task is the number of line segments overlapping the CMBR of a polygon multiplied by the number of line segments overlapping the CMBR of another polygon. For the definition of the workload, we usedthe symbols as described in Table 1.

After using PolySketch: $W_{PS} = C * \frac{P}{T_P} . \frac{Q}{T_Q}$

After using PSCMBR: $W_{PSCMBR} = \hat{C} . \hat{P} . \hat{Q}$

To illustrate the workload calculation, suppose we have two tiles of a polygon, where one tile overlaps with three tiles and another tile overlaps with 10 tiles of another polygon. Assuming the tile sizes for both polygons are 5, the refinement workload for this task is 1*3*5+1*10*5=65.

Table 4, 5 and 6 show the performance of PSCMBR for three real datasets. Workload means the actual computational workload in LSI function. To show the filter efficiency on top of standard filtering using R-tree, the percentage of candidate tasks discarded is calculated using candidates produced by R-tree as a baseline. We subtract the number of candidates produced by R-tree with the remaining number of candidates after using a given filter and then divide the difference by the baseline. We do not need to perform further refinement on the discarded tasks. Candidate tile pairs need further refinement using LSI function. Run-time is expressed in seconds and it includes execution time for the filter and refine phases for the real datasets. For the run-time of CMF, we show two execution time results. The first number is the time of checking and storing line segments overlapping CMBR on CPU. The second number is the time of only refinement phase using LSI function on GPU after applying CMF.

Table 4. Different filters effect on the LSI function for Water dataset

Water	Workload	Candidate Tasks Discarded	Candidate tile pairs	Run-time (s)
CMF	16,327,012,938	73.13%	NA	10.36+4.53
PS	1,789,226,826	68.48%	18,792,164	1.39
PSC-MBR	154,187,055	75.46%	17,417,707	0.62

Table 5. Different filters effect on the LSI function for Urban dataset

Urban	Workload	Candidate Tasks Discarded	Candidate tile pairs	Run-time(s)
CMF	25,737,640	71.53%	NA	0.23+0.03
PS	7,489,801	66.09%	152,219	0.06
PSC-MBR	540,240	72.83%	100,052	0.02

According to the Table 4, 5 and 6, we can see PSCMBR can reduce much more workload in LSI function for all three datasets when compared to the other two filters. After using PSCMBR, the LSI workload is 99.1%, 97.9% and 98.1% smaller than using CMF for Water, Urban and Lake datasets. The LSI function workload is 91.4%, 92.8% and 86.7% smaller than using PolySketch for the three datasets. PSCMBR combines the strength of CMF and PolySketch so it can handle more general cases. It can also discard the line segments which are inside the same tile. PSCMBR can also discard more candidate tasks in total compared to other filters so there are fewer candidate tasks that need refinement phase. In addition, the number of candidate tile pairs after using PSCMBR is on average 29.73% smaller than using PolySketch.

Table 6. Different filters effect on the LSI function for Lake dataset

Lake	Workload	Candidate Tasks Discarded	Candidate tile pairs	Run-time(s)
CMF	260,210,378	80.81%	NA	9.4+0.51
PS	37,464,000	70.96%	1,286,389	1.17
PSC-MBR	4,972,603	82.21%	674,667	0.80

In the first step, PSCMBR discards more candidate tasks which do not need further refinement. In the second step, it discards more candidate tile pairs and it reduces the false hits which do not need further refinement. According to Table 6, it reduces up to 47.6% candidate tile pairs by using PSCMBR instead of PolySketch. For the run-time also, we can see PSCMBR works well. Compared to PolySketch, PSCMBR filter yielded 2.24X, 3X, and 1.46X speedup for Water, Urban and Lake datasets. Even if we only compare the refinement time of PSCMBR with LSI function to the refinement time with LSI function after using CMF, PSCMBR also works well. The size of Lake dataset is huge which leads to higher overhead of copying the Lake data from CPU memory to the GPU memory for PSCMBR filter. However, the data copy overhead is lower for CMF because we do pre-processing on CPU, so the size of data copied to GPU is much smaller because it only contains the line segments overlapping the CMBRs. This explains why PSCMBR-based refinement with LSI function is little slower than the refinement time of LSI function after using CMF.

5.4 Results for different PSCMBR tile-sizes

Table 7. Performance variation while using different tile-sizes for Water dataset

Tile-size	Current Workload	Candidate Tasks	Run-time(s)
15-5	139,698,900	250,064	0.715
15	154,158,443	258,703	0.719
20	178,527,782	261,640	0.671
20-10	164,191,106	256,226	0.627
30	232,956,052	265,858	0.644
30-10	198,688,474	257,191	0.627
40	292,387,187	269,139	0.670
50	355,278,337	272,016	0.726

The real-world datasets are complicated since it contains different sizes of polygons. Therefore, the tile-size used in the first step for filtering is a factor that affects the performance. Table 7 shows the performance of using different tile-size for Water dataset. Similar to PolySketch, we can either use one tile-size for all polygons or use two tile-sizes for different polygons. In Table 7, for some rows, there are two numbers in 'tile-size' column. The first and second numbers are the tile sizes for large and small polygons. In the experiments, if we use two tile-sizes for the polygons, we use larger tile-size for the large polygons (with more than 400 vertices) and smaller tile-size for the small polygons (with less than 400 vertices). We found that smaller tile-sizes perform better in our prior work [11]. We use different tile-sizes for Water dataset to test the performance of PSCMBR.

According to Table 7, we can see that the range of tile-sizes that can be chosen is large since we can get similar run-time results by setting tile-size as 20, 30 or 40. Although the current workload is increasing, the run-time results are similar. Using two tile-sizes at the same time can reduce more workload compared to only using one tile size. Run-time results are also better. In addition, using smaller tile-size can reduce more workload in LSI function.

5.5 System Performance with PNP Filters

Table 8 shows the whole system run-time results. To be fair, we compare them with the results of using one GPU. We can see the performance of the whole system is much improved. The new system gets 6.36X and 9.56X speedup compared to HiFiRe system for the Urban and Water data sets. One reason is that we use GPU to pre-process data for PNP test instead of CPU. In our improved system equipped with new PNP filters, we do not need to store the line segments and vertices for PNP test because we can make full use of the tiles used in LSI function and do more calculations within the PolySketch-LSI function to get the information that will be used in PNP test. This also avoids using more memory and data movement between CPU and GPU. Another reason is that the new algorithm can handle different types of polygons, such as very small, medium or huge polygons.

Table 8. End-to-end run-time (does not include R-tree time)

	HiFiRe running time(s) [11]	New HiFiRe running time(s)
Urban	0.35	0.055
Water	10.63	1.109

5.6 Filters with PNP TestWorkload

To show the efficiency, we compare PolySketch-based PNP function with Stripe-based PNP function (using 8 stripes). We also compare tile-based PNP function with constant vertex PNP function. For the workload of PNP test using polygons of size P and Q, every vertex from A1 should be compared with all line segments from B1 and every vertex from B1 should be compared with all line segments from A1. Therefore, the workload is $2 * P * Q$ for every task. In addition, the total workload for PNP test is the summation of workload of individual tasks. Table 9 is about the workload of the tasks where two polygons have line segment intersections. Since the PolySketch-based PNP workload in each polygon of the same task are different, we also update the workload of stripe-based PNP [11]. We can see that the workload is still much reduced by using PolySketch-based PNP function even compared with Stripe-based PNP test. For Urban and Water, it reduces 41.2% and 79.2% of the workload of Stripe-based PNP function.

Table 9. Workload using different methods in tasks where two polygons have line segment intersections.

	Stripe-based PNP workload	PolySketch-based PNP workload
Urban	28,642,336	16,854,370
Water	10,602,276,252	2,200,221,374

PolySketch-based PNP function classifies the tiles into two categories, namely, $intersection\ tile$ and $no\ intersection\ tile$. Figure 8 shows the percentage of how many tiles are considered as the intersection tile and no-intersection tile. We can see 96.8% and 97.7% tiles are considered as $no\ intersection\ tile$ for Urban and Water. This definitely reduces the workload and increases the efficiency of PNP filter.

For the $intersection\ tile$, we can see the percentage is 3.2% and 2.3%. Although we have to do PNP test for all vertices within intersection tiles, the total number of such tiles is not large. In addition, PolySketch-based PNP function can still reduce more workload for these tiles because we compare a test vertex only with the line segments within the tiles whose MBR overlaps with the horizontal ray passing through the test vertex by considering y-coordinate.

According to Table 10, we can see tile-based PNP function can also reduce the workload. For Urban and Water, it can reduce 98.8% and 98.6% of the workload when compared to the constant vertex PNP workload. The advantage is that we can keep the constant vertex PNP test's strengths and discard the line segments within the tiles which can not overlap with test vertex by only considering y-coordinate.

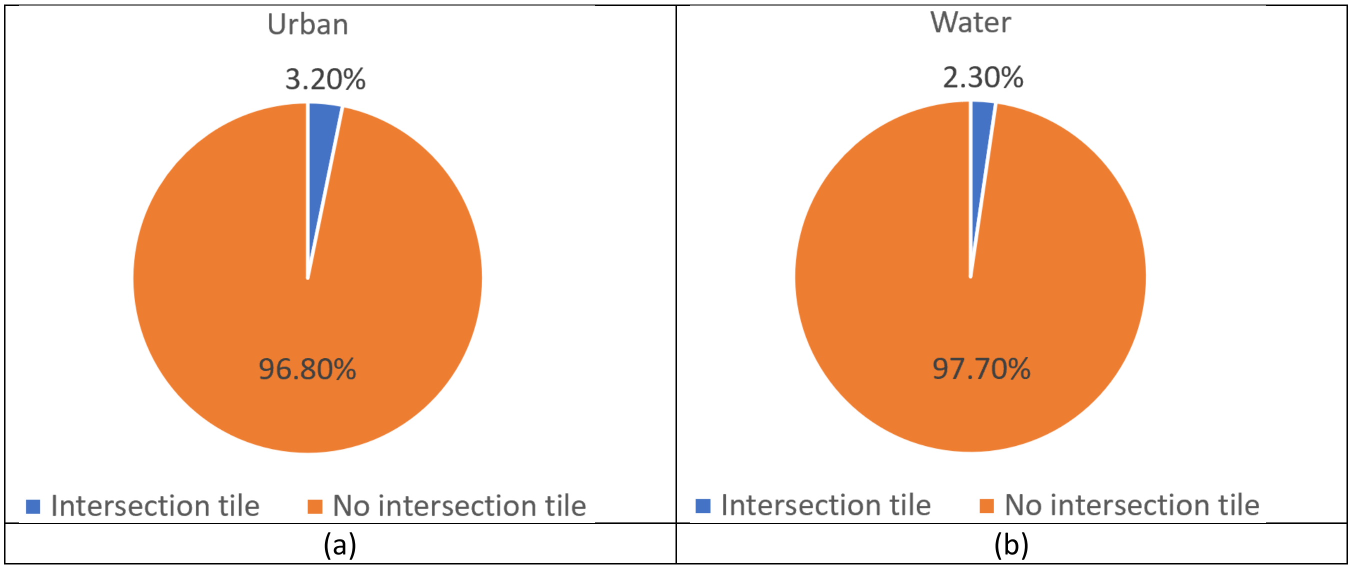

Fig. 8. Percentage of $Intersection\ tile$ and $no\ intersection\ tile$ for tasks where two polygons have line segment intersections

Table 10. The workload in PNP test for the tasks where one polygon may be totally inside another polygon

	Constant vertex PNP workload [11]	Tile-based PNP workload
Urban	277,495,510	3,452,066
Water	10,352,636,305	145,693,382

5.7 New Hierarchical Filter and Refine System

For this new filter and refine system, since we also use R-tree to index input datasets, the total overlay processing time should also include the time of using R-tree. For Water and Urban datasets, the time taken by R-tree filter on CPU is 2.27s and 0.065s. Therefore, the end-to-end time of the new HiFiRe System is 3.379s and 0.12s.

To show the performance of PSCMBR HiFiRe system, we define the processing rate:

Processing rate = Input line segments/Overlay processing time.

For Water dataset, the number of line segments in layer 1 and layer 2 are 24,739,074 and 60,305,435. Therefore, the number of input line segments is 85044509. The processing rate is 77 million/sec. For Urban dataset, the number of line segments in layer 1 and layer 2 are 1,153,348 and 1,332,830. Therefore, the number of input line segments is 2,486,178. The processing rate is 45 million/sec.

6 Conclusion

We have developed new filters used in filter and refine technique and demonstrated the benefits in our improved HiFiRe system. The new filters make geometric intersection computations faster on a GPU. Compared to CMF, the new PSCMBR filter can efficiently handle the case where the CMBR of two polygons is large. Compared to PolySketch, the new filter is more efficient in minimizing the false hits and decreases the workload in the refinement phase. For line segment reporting and point-in-polygon tests inherent in spatial join and polygon overlay algorithms, we have shown considerable workload reduction and better run-time using a GPU accelerator. Moreover, our PNP filter leverages PolySketch and this has resulted in significant end-to-end performance improvement in HiFiRe system.

REFERENCES

[1] Danial Aghajarian and Sushil K Prasad. 2017. A spatial join algorithm based on a non-uniform grid technique over GPGPU. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 56.

[2] Danial Aghajarian, Satish Puri, and Sushil Prasad. 2016. GCMF: an efficient end-to-end spatial join system over large polygonal datasets on GPGPU platform. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 18.

[3] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. Robust and efficient polygon overlay on parallel stream processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 304–313.

[4] Wael M Badawy and Walid G Aref. 1999. On local heuristics to speed up polygon-polygon intersection tests. In *Proceedings of the 7th ACM international symposium on Advances in geographic information systems*. 97–102.

[5] Thomas Brinkhoff, H-P Kriegel, and Ralf Schneider. 1993. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 40–49.

[6] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1994. Multi-step processing of spatial joins. *Acm Sigmod Record* 23, 2 (1994), 197–208.

[7] Erich L Foster, Kai Hormann, and Romeo Traian Popa. 2019. Clipping Simple Polygons with Degenerate Intersections. *Computers & Graphics*: X (2019), 100007.

[8] Chao Gao, Furqan Baig, Hoang Vo, Yangyang Zhu, and Fusheng Wang. 2018. Accelerating Cross-Matching Operation of Geospatial Datasets using a CPU-GPU Hybrid Platform. In *2018 IEEE International Conference on Big Data (Big Data).* IEEE, 3402–3411.

[9] Günther Greiner and Kai Hormann. 1998. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics (TOG)* 17, 2 (1998), 71–83.

[10] Edwin H Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems (TODS)* 32, 1 (2007), 7–es.

[11] Yiming Liu, Jie Yang, and Satish Puri. 2019. Hierarchical Filter and Refinement System Over Large Polygonal Datasets on CPU-GPU. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC).* IEEE, 141–151.

[12] Salles VG Magalhães, Marcus VA Andrade, W Randolph Franklin, and Wenli Li. 2015. Fast exact parallel map overlay using a two-level uniform grid. In *Proceedings of the 4th International ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. 45–54.

[13] David M Mount. 1997. Geometric intersection. In *Handbook of Discrete and Computational Geometry*, chapter 33. Citeseer.

[14] Jürg Nievergelt and Franco P. Preparata. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM* 25, 10 (1982), 739–747.

[15] Anmol Paudel and Satish Puri. 2018. OpenACC Based GPU Parallelization of Plane Sweep Algorithm for Geometric Intersection. In *International Workshop on Accelerator Programming Using Directives*. Springer, 114–135.

[16] Satish Puri and Sushil K Prasad. 2015. A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system using mpi. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 576–585.

[17] Michael Ian Shamos and Dan Hoey. 1976. Geometric intersection problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976).* IEEE, 208–215.

[18] Darius Sidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. 2018. Improving Spatial Data Processing by Clipping Minimum Bounding Boxes. In *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE, 425–436.

[19] Peter van Oosterom. 1994. An R-tree based map-overlay algorithm. In *Proc. EGIS*, Vol. 94. 318–327.

[20] Hein Veenhof, Peter Apers, and Maurice Houtsma. 1995. Optimisation of Spatial Joins Using Filters. In *Advances in Databases, 13th British National Conference on Databases*, Manchester, United Kingdom. Springer, 136–154.

[21] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H Saltz. 2012. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 5. NIH Public Access, 1543.

[22] Simin You, Jianting Zhang, and Le Gruenwald. 2016. High-performance polyline intersection based spatial join on GPU-accelerated clusters. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 42–49.

[23] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T Silva, and Juliana Freiref. 2017. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *Proceedings of the VLDB Endowment* 11, 3 (2017), 352–365.

[24] Jianting Zhang and Simin You. 2012. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 23–32.

Notes

[1]https://rogue-modron.blogspot.com/2011/04/polygon-clipping-wrapper-benchmark.html

[2]https://www.naturalearthdata.com/downloads/10m-physical-vectors/10m-ocean/