GCMF: An Efficient End-to-End Spatial Join System over Large Polygonal Datasets on GPGPU Platform *

Danial Aghajarian
Computer Science Dept.
Georgia State University
daghajarian@cs.gsu.edu

Satish Puri
Mathematics, Statistics, and
Computer Science Dept.
Marquette University
satish.puri@marquette.edu

Sushil Prasad
Computer Science Dept.
Georgia State University
sprasad@gsu.edu

ABSTRACT

Given two layers of large polygonal datasets, detecting those pairs of cross-layer polygons which satisfy a join predicate, such as intersection or contain, is one of the most computationally intensive primitive operations in the spatial domain applications. In this work, we introduce *GCMF*, an end-to-end software system, that is able to handle spatial join (with *ST_Intersect* operation) over non-indexed polygonal datasets with over 3 GB file size comprising more than $600,000$ polygons on a single GPU within less than 8 sec by applying innovative filter and refinement techniques. *GCMF* performs a two-step filtering phase. 1) A sort-based Minimum Bounding Rectangle (MBR) filtering step detects potentially overlapping polygon pairs up to 20 times faster than the optimized GEOS library routine. 2) A linear time Common MBR filtering step (based on the overlapping area of two given MBRs) that not only eliminates two-third of the candidate polygon pairs but also reduces the number of edges to be considered in the refinement phase by 40-fold on an average based on our experimental results with real datasets. Furthermore, for the refinement phase, *GCMF* implements a load-balanced parallel *point-in-polygon* and *edge-intersection* tests over GPU. Our experimental results with three different real datasets show up to 39-fold end-to-end speedup versus optimized sequential routines of GEOS C++ library as well as PostgreSQL spatial database with PostGIS.

Categories and Subject Descriptors

H.4 [**Information systems**]: Geographic information systems; F.2 [**Theory of computation**]: Shared memory algorithms

Keywords

Spatial join, HPC, Parallel algorithm, GPGPU

*This research is partially supported by NSF grant #1205650.

SIGSPATIAL'16, October 31-November 03, 2016, Burlingame, CA, USA

DOI: http://dx.doi.org/10.1145/2996913.2996982

1 Introduction

Spatial data comprising rectangles, polygons, lines, and points are wide-spread in Geographic Information Systems (GIS). Because of advanced remote sensing technologies, the volume of data generated in such applications has tremendously increased over the past decade. For instance, Light Detection and Ranging (LiDAR) systems produced 40 PB of data in 2014 but domain scientists were able to handle only $30PB$ of data [4]. Researchers have predicted LiDAR technology will generate up to $1,200$ PB of data by 2020, while GIS workforce has the capacity to process only 50 PB. This demonstrates ever-increasing demand for High Performance Computing (HPC) in GIS domains. In particular, GPUs are very popular among the HPC technologies as they are widely available at low prices yet with powerful features. For example, GeForce GTX 1080, the latest NVIDIA GPU released in May 2016, is driven by the new NVIDIA Pascal$^{\text{TM}}$architecture which provides 2560 *Cuda Cores* operating at 1607 MHz base clock and 8 GB of the main memory with 320 GB/sec bandwidth that makes it feasible to handle larger data in a real time manner.

Spatial join is one of the most computationally intensive operations in spatial computing. For instance, spatial join of a polyline table with 73M records representing the contiguous USA with itself takes roughly 20 hours to complete on an Amazon EC2 instance [14]. Therefore, harnessing parallel processing capabilities of modern hardware platforms to perform join operation over big spatial datasets is essential. In general, spatial join can be defined as follows: given two spatial datasets R and S and a spatial join predicate $\bowtie$ (e.g., overlap, contain, intersect) as input, spatial join returns the set of all pairs (r, s) where $r \in R$, $s \in S$, and $\bowtie$ is true for (r, s) [6]. A typical application of a spatial join is "Find all pairs of rivers and cities that intersect." The focus of this paper is on polygonal data with *ST_Intersect* operation in which for a given pair of polygons, it returns true if and only if polygons share any portion of space [12].

Generally, spatial join algorithms over polygonal data follow a two-phase paradigm [6]:

- Filtering phase: reduces all the possible cross-layer polygon pairs to a set of potentially intersecting candidate pairs based on minimum bounding rectangle overlap-test.

- Refinement phase: removes any results produced during the filtering phase that do not satisfy the join condition.

The refinement phase is significantly time-consuming. For instance, an analysis of join operation on CPU over more than 10,000 spatial objects in [1] shows that refinement phase takes five times more than the rest of the operations including filtering and parsing datasets. While this study demonstrates the significance of refinement step, in the current literature, most GPU-related works have only addressed the filtering phase algorithms. In this work, we plan to bridge this gap by introducing *GCMF*, a GPU-based spatial join system including both filtering and refinement steps. Our work can be distinguished in two ways: 1) To the best of our knowledge, there is no such system to process end-to-end polygonal intersection-based join over GPU, and, 2) comparable systems that proposed other spatial join predicates such as k-Nearest Neighbor [8] cannot handle the amount of data that we are able to process on a single GPU and they have reported less speedups.

In summary, our key contributions in this work are:

- *GCMF*: An end-to-end spatial join system built on a single GPU to generate cross-layer polygon pairs from two large datasets that satisfy spatial join condition in near real-time manner. Based on our experimental results, *GCMF* was able to handle real datasets as large as (not limited to) $3GB$ files up to 39 times faster than an *optimized GEOS* library within a few seconds.

- A sort-based MBR filtering algorithm with a suitable GPU-specific data structure that yields up to 20-fold speedup compared to *optimized GEOS* library. Proof of correctness of this algorithm is also provided.

- Common MBR Filter (CMF) based on the MBR resulting from the intersection of MBR_{P1} and MBR_{P2} that makes *refinement* phase 28 times faster than the same implementation without *CMF* by 1) reducing the number of candidate polygon pairs by up to 66% and 2) making the polygon pairs 40-fold smaller in size by removing many of the non-intersecting edges. We provide proofs and performance analysis for this filter.

- A load-balanced implementation of parallel *point-in-polygon* test that is up to 9 times faster compared to the naive implementation over GPU. It also achieves 30-fold speedup compared to sequential implementation over CPU.

The remainder of this paper is organized as follows. In the next section, we summarize the current work in the literature with a focus on performance of algorithms. Then, in Section 3, we introduce *GCMF* system overview, its components, algorithms and theoretical analysis. Experimental results are presented in Section 4. Finally, we provide conclusions and point out our future work plan.

2 Literature Review

Exploiting GPU to do spatial join operations has been explored in the HPC literature [20, 21, 13, 9, 23]. In this section, we have summarized related work focusing on the design and implementation of various types of join operations. First, we present a brief survey on sequential and multi-core algorithms. Then, we briefly summarize the work around GPU algorithms for spatial join.

Sequential and multi-core spatial joins algorithms: A sequential plane-sweep MBR filtering algorithm has been explained in [2]. In this paper, the basic idea is to sort lower boundaries of rectangles for plane sweeping in a recursive manner. An extensive performance evaluation on synthetic datasets with various ranges of properties including tall-shape or wide-shape rectangles is carried out in this paper. The results shows that the algorithm efficiency significantly depends on the width to height ratio of the rectangles. Some methods for dealing with tall/wide rectangles is described in [5]. GIPSY [11] is a novel approach for spatial join of two datasets with contrasting density to address space oriented coarse-grained partitioning challenges. GIPSY partitions the dense dataset using a method similar to STR [7] and then joins it with the non-indexed sparse dataset. Their extensive evaluation results using synthetic and real datasets yields up to 18-fold speedup. The main limitation of this algorithm is the assumption that one of the datasets is sparse.

A bottom-up spatial join approach based on CPU parallelism has been proposed in [26]. This algorithm does not rely on pre-existing spatial indices. The MBR join over Sequoia2000 dataset [19] (58411 by 20,974 size) takes more than 7 seconds on a platform with 40 processors.

GPU-based spatial joins algorithms: A naive parallel implementation of spatial join using R-tree has been described in [20]. This top-down query search method runs about 3 times faster on GPU than CPU on average and considering the CPU-GPU data transfer time, the performance is even worse than CPU implementation. Parallel spatial join using R-tree has been implemented in [21]. The GPU algorithm runs 8 times faster than multi core CPU implementation. A simple parallel r-tree query implementation on GPU is stated in [9] which runs 20 times faster than CPU. Another R-tree-based spatial join on GPU with less than 4-fold speedup has been reported in [20]. In [18], six spatial join queries has been implemented over GPU and they have achieved 6-10 fold speedup including transfer time from CPU to GPU. One of the fastest R-tree implementation and querying on GPU has been recently described by our group in [13]. We have proposed five algorithms for batch MBR querying and the best performance comes out of the modified-DFS algorithm, which initiates all-to-all search starting from the parents of the R-tree leaves. Speedup gain for querying algorithms is in the range 76-fold to 153-fold which is much higher than previous algorithms in the literature. However, a key limitation is the small datasets it currently handles because of $O(n^2)$ space complexity.

Spatial Join Processing Systems:

CudaGIS [23] exhibits 20-40 fold speedup versus sequential CPU implementation for spatial indexing and some spatial join operations. This group has also exploited the uniform grid-based approach to create various indexing data structures such as R-tree, quad-tree, CSPT-P-tree and BMMQ-tree data structures. [25]. The main idea is to assign rectangles to grid cells and compute the operations locally in each grid cell. The speedup results show up to 20-fold improvement over CPU implementation for end-to-end system. Although the results show a good speedup over various spatial operations, this library has some limitations. As opposed to spatial join described in [24] which is ST_Within operation on a point layer and polygon layer, our work is $ST\_Intersect$ on two polygon layers. Also, most of their experiments are

done over the datasets with an assumption of relatively uniform load per thread due to small size polygons.

An Impala-based in-memory Spatial Processing system has been designed in [22]. Most of their library has been implemented on top of Thrust parallel library in Cuda SDK. Single node performance test of their framework demonstrates speedup less than 2 for two different datasets.

3 Algorithms

In this section, we present the problem definition and notations, datasets employed, the overall system design, and the filtering and refinement algorithms.

3.1 Problem Definition

Given a polygon P, $MBR_P = (x_{P,0},\ y_{P,0},\ x_{P,1},\ y_{P,1})$ is the minimum bounding rectangle of P that can be described by its bottom-left coordinate $(x_{P,0},\ y_{P,0})$ and top-right coordinate $(x_{P,1},\ y_{P,1})$. We also use x_P (or y_P) to refer to x-coordinates of MBR_P regardless of being left or right coordinate ($x_P \in \{x_{P,0},\ x_{P,1}\}$). For two overlapping bounding rectangles, MBR_{P1} and MBR_{P2}, we define *Common MBR*, $MBR_{P1 \cap P2}$, as the minimum bounding rectangle of their overlapping area. Finally, for any polygon P, E_P is the list of edges and $E_P(i)$ denotes i-th edge.

As stated before, spatial join operation can be defined over two spatial objects and a predicate. In this paper, we define spatial join as follows: for any given pair of polygons, $P1$ and $P2$, $P1 \bowtie P2$ returns true, if and only if either there exists a pair of edges $E_{P1}(i)$ and $E_{P2}(j)$ such that they intersect, or if overlap or one of the polygons lies inside the other one.

3.2 Datasets

We have used two real polygonal dataset pairs (*Urban*, *Water*) from http://www.naturalearthdata.com and http://resources.arcgis.com from GIS domain with various sizes and characteristics (Urban and Water). The third dataset (*Telecom*) comes from telecommunication domain. The details of the datasets are provided in Table 1. All the datasets are available online at the project site at http://grid.cs.gsu.edu/~daghajarian1/SpatialJoin.html in both shapefile and text formats.

Table 1: Three real datasets used in our experiments.

Label	Dataset	Polygons	Size
Urban	*ne_10m_admin_states*	11,878	$46MB$
	ne_10m_urban_areas	4,646	$41MB$
Telecom	*GA_telecom_base*	101,860	$171MB$
	GA_telecom_overlay	128,683	$240MB$
Water	*US_block_boundaries*	219,831	$2.175GB$
	US_water_bodies	463,591	$921MB$

3.3 System Design Overview

ST_Intersect predicate requires both *edge-intersection* and *point-in-polygon* tests. Figure 1 illustrates a typical workflow of the spatial join algorithm which has been used in the literature. R-trees are used to index polygons and then R-tree query is used to detect potentially overlapping polygons. Finally, *point-in-polygon* and *edge-intersection* tests are applied in the refinement phase. Overall running time of the traditional system is heavily dominated by the refinement phase which we try to address by introducing our new system design.

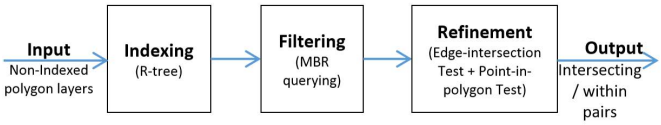

Figure 1: Typical spatial join processing pipeline

Edge-intersection test is more compute-intensive than *point-in-polygon* test. We take advantage of this fact in our system design workflow by adding one more filtering phase based on Common MBR. Figure 2 shows the overview of the *GCMF* system. *GCMF* has two subsystems. The first subsystem includes two filtering components. The first is Sort-based MBR Filter (*SMF*) which reduces set of all cross-layer polygon pairs into the set of potentially intersecting polygon pairs (**C**) by overlap-test over their minimum bounding rectangles. The second component is Common MBR Filter (*CMF*) that applies intersection test to edges of each pair in **C** and their common MBR to classify polygon pairs into following three groups: 1) *Intersecting-Edge candidate set* (**I**), 2) *within candidate set* (**W**) and 3) disjoint pairs which can be discarded. We explain these two filters in more details in the following subsections. The refinement subsystem comprises two components: *point-in-polygon* test (*PnP_Test*) and *edge-intersection* test (*EI_Test*). The first component takes **W** as input and performs the *point-in-polygon* test. If a pair passes the test successfully, it goes to output directly, otherwise it is sent to *edge-intersection* test for further processing. As shown in the Figure 2, the input of the *EI_Test* comes from **I** as well as those pairs from **W** which failed *point-in-polygon* test. Finally, *EI_Test* adds a pair to output if it can detect at least one cross-layer edge-intersection/overlap in that pair.

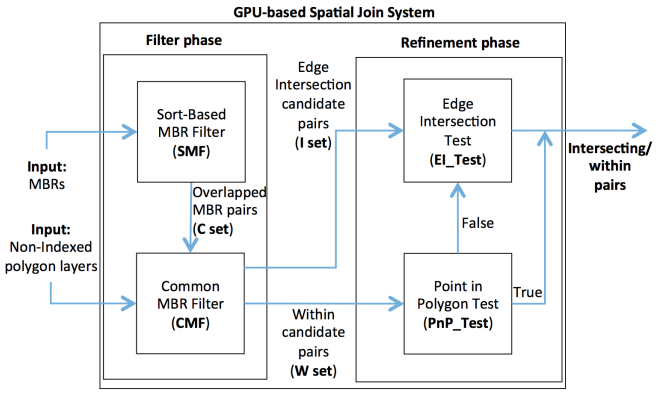

Figure 2: GCMF system design overview

3.4 Sort-based MBR Filtering

Tree-based data structures such as R-trees and interval trees have been used in MBR filtering. While these data structures are perfectly matched to sequential algorithms, they are not suitable currently for large datasets over GPUs mostly because of their hierarchical structures and memory usage

in current implementations. To address this issue, we introduce *SMF* which is a sort-based MBR filter algorithm, highly suitable for GPUs in particular.

SMF takes *MBR* sets **R** and **S** with $|\mathbf{R}| = m$ and $|\mathbf{S}| = n$ as input and generates cross-layer MBR-overlapping pairs as output set **C**. If two MBRs overlap then their (interval defined by) x-coordinates overlap and their y-coordinates overlap. In essence, this algorithm looks for two interval overlaps in x and y dimensions. It sorts the x coordinates of the MBRs from both layers (set **X**). Then for each MBR_i with x coordinates $(x_{i,0}, x_{i,1})$, it finds all the MBR_j from the other layer with x-coordinates $(x_{j,0}, x_{j,1})$ such that $x_{i,0} \leq x_{j,0} \leq x_{i,1}$. Then MBR_i is tested against all such MBR_j for overlap in their y coordinates, thus yielding the output set C. Same can be done by sorting the y coordinates and then testing in x-dimension. Later, we prove that this algorithm neither generates a duplicate pair nor misses one.

Algorithm 1 describes the Sort-based MBR Filter suitable for GPUs. As mentioned above, **X** is a vector of x-coordinates of the MBRs in both layers. *CRadixSort* is our customized radix sort function which generates two vectors:

- *sortIndex*: In order to prevent swapping 64-bytes elements in **X** over GPU main memory which is not efficient, CRadixSort prepares sorted indices such that *sortIndex[i]* is the index of i-th smallest element in **X**.
- *rankIndex*: To have an efficient parallel algorithm, each MBR_i needs to know indices of its left and right coordinates in vector **X** in $O(1)$ time without searching through *sortIndex*. To provide this information, we introduce *rankIndex* which keeps track of MBRs in *sortIndex* vector. *rankIndex[i]* is the index of x_i at *sortIndex*.

The following properties are always held by these two vectors for any $0 \leq i \leq m+n-1$:

$$\begin{aligned} rankIndex[\, sortIndex[\, i\,]\,] &= i \\ sortIndex[\, rankIndex[\, i\,]\,] &= i \end{aligned} \tag{1}$$

To better understand the data structure, Figure 3 provides an example. Part (b) is **X** for 4 MBRs presented in part (a). For any MBR_i, $x_{i,0}$ and $x_{i,1}$ can be accessed through $2 \times i$ and $2 \times i + 1$ indices of **X** respectively. For instance, the first two values are left and right x-coordinates of MBR_0. Parts (d) and (e) represent *sortIndex* and *rankIndex*, respectively.

Generating *rankIndex* has two advantages. 1) For a given MBR_i, we can access its sorted indices in O(1). For example, position of $x_{3,0}$ at *sortIndex* (equal to 1) is the 6th element of *rankIndex*. 2) This vector is helpful for balancing the load. If we want to figure out how many elements may potentially lie in between an MBR range (an estimation of load of the block handing that MBR), we can subtract its corresponding values in the *rankIndex* vector. The value gives us an upper bound which also can be used as a relative measure of number of overlapping MBRs for a given MBR. For example, MBR_3 includes just one element in its x-interval $\{x_{0,0}\}$, while MBR_1 has three $\{x_{3,0}, x_{0,0}, x_{3,1}\}$.

To implement sort-based MBR filter, we launch a kernel with $m+n$ (total number of MBRs in both layers) blocks and each block handles the interval corresponding to one MBR. By evenly distributing the load among the threads within a block, we make the implementation load-balanced. In GPU $Block_i$, algorithm finds all the elements $x_{j,0} \in X$ which lie between $x_{i,0}$ and $x_{i,1}$ using *rankIndex* and *sortIndex* (Line 5 in Algorithm 1). Then if they also intersect in y-coordinate, the block produces (i, j) pair as output if MBR_i is the first-layer MBR (Line 8), otherwise, (j, i) is generated (Line 10).

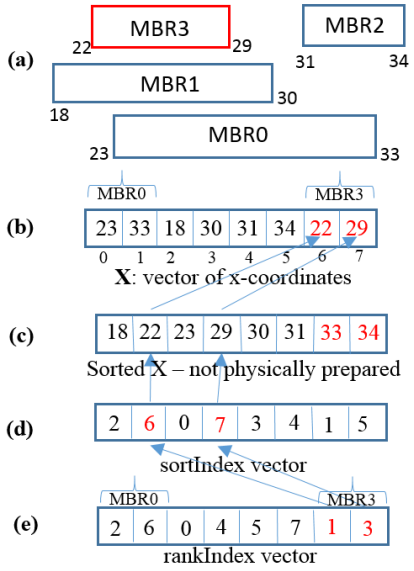

Figure 3: An example of data structure used for sort-based MBR filter. Part (a) is actual MBRs, (b) is X set, (c) is sorted X, (d) is sorted indices of X and (e) is sorted indices of MBRs. For example index of 2-th smallest coordinate (which is 22) can be retrieved from $sortIndex[1] = 6$**. Part (e) is** *rankIndex* **which keeps track of MBRs in** *sortIndex*. **For example, position of** $(x_{3_0}, x_{3,1})$ **(red MBR)in** *sortIndex* **can be fetched from** $rankIndex[3 \times 2] = 1$ **and** $rankIndex[3 \times 2 + 1] = 3$

Lemma 1 proves the correctness of sort-based MBR filtering algorithm.

LEMMA 1. ***SMF**: Given two sets of MBRs, **R** and **S**, Algorithm 1 will generate all overlapping MBR pairs without any false positives or duplicates.*

PROOF. The first part of proof can be derived from the algorithm by showing that $\mathbf{O} \subseteq \mathbf{A}$ and $\mathbf{A} \subseteq \mathbf{O}$ where **O** is output of algorithm and A is pairs of overlapping MBRs. For the second part, lets assume MBR pair (i, j) is generated twice. Pair (i, j) may be generated by $Block_i$ or $Block_j$. As such, one of these two blocks may generate this pair twice or each of the blocks may generate only one of the duplicated pairs. First case is impossible as we process each MBR just once in its corresponding block and no coordinate is duplicated in the data structures. Let us assume both $Block_i$ and $Block_j$ reports (i, j) as output. Given this, $Block_i$ implies that $x_{j,0}$ appears after $x_{i,0}$ and before $x_{i,1}$ in **X**. $Block_j$ also requires that $x_{i,0}$ appears between $x_{j,0}$ and $x_{j,1}$ in **X** at the same time which is impossible. Therefore, each intersecting pair (i, j) is exactly generated once in the output. □

3.4.1 SMF Analysis

For average case analysis, we assume that average height and width of MBRs are $\bar{h}$ and $\bar{w}$ respectively and they are

Algorithm 1 Sort-based MBR filtering algorithm

Input: **R** and **S** set of MBRs **Output**: set **C**

Building data structure
let $\mathbf{X} = \{x_i | x_i \in x\text{-}coordinate\ of\ \mathbf{R} \cup \mathbf{S}\}$
$(sortIndex\ ,\ rankIndex) \leftarrow$ CRadixSort($\mathbf{X}$)

procedure (FILTER FOR **R** AND **S** MBRS)
  for each $GPU\ Block_i$, $0 \leq i < (m+n)$, **do in parallel**
    for each $x_{j,0}$, $x_{i,0} \leq x_{j,0} \leq x_{i,1}$ **do**
      if $(y_{j,0}, y_{j,1})$ *intersects* $(y_{i,0}, y_{i,1})$ **then**
        if $MBR_j \in S$ **then**
          Add pair (i, j) to the output set
        else
          Add pair (j, i) to the output set
        end if
      end if
    end for
  end for
end procedure

scattered in a $H_a \times W_a$ rectangle area. The sequential time complexity of *CRadixSort* is $O(n \cdot b)$ where b is the average number of digits of the coordinate values. Also, Algorithm 1 includes two nested loops. The outer loop has $n+m$ iterations. The inner loop goes through all the x-coordinates lying in the range $(rankIndex(x_{i,0}), rankIndex(x_{i,1}))$ for a given MBR_i which on the average has d_{avg} elements ($\bar{w}/W_a$ fraction of (n+m) MBRs).

$$d_{avg} = \frac{\bar{w} \times (n+m)}{W_a} \tag{2}$$

Thus, the algorithm's sequential complexity is:

$$= O((n+m) \cdot b + \frac{\bar{w}}{W_a} \cdot (n+m)^2) \tag{3}$$

The complexity of the algorithm depends on $\frac{\bar{w}}{W_a}$ factor which is proportional to the number of output pairs. If $\frac{\bar{w}}{W_a} = O(\frac{1}{n+m})$ then the second term in equation 3 becomes linear and therefore total complexity becomes $O((n+m) \times b)$. This is what usually happens in real datasets. If we have $\frac{\bar{w}}{W_a} = O(1)$, the complexity would be order of $O((n+m)^2)$. One of the scenarios that may lead to $O((n+m)^2)$ complexity, is the case with $\bar{w} \approx W_a$ which means each MBR is almost as wide as the entire area and therefore has potential overlap with almost all the other MBRs.

SMF has linear space complexity. As described in the preamble of Section 3.4, it estimates the maximum number of overlapping MBRs for a given MBR and then allocates the memory in advance. Although this estimation has some time-overhead, applying this strategy makes it possible for *SMF* to test large MBR layers for intersection. *SMF* could process two datasets each including more than $1M$ MBRs on a GPU node with 6 GB of the main memory.

3.4.2 SMF Performance

We have used a sequential optimized *GEOS* library as baseline to compare with *SMF*. Table 2 shows our experimental results using all three datasets introduced in Section 3.2.

The results in this table shows up to 20-fold speedup. As we explained in the related work, RTree method introduced in [13] can achieve more speedup versus *SMF*, but it cannot query datasets with more than around 20,000 MBRs in the second layer because the memory space requirement of its current implementation is $O(n \cdot m)$ due to matrix based data structure for $O(1)$ access.

Table 2: Running time of *SMF* and GEOS for MBR filtering

Dataset	Running time (ms)		♯ of Outputs
	GEOS	SMF	
Urban	197	16	28,687
Telecom	2,683	240	747,086
Water	13,048	676	1,020,458

Also to show the scalability, Figure 4 plots *transfer* time and *SMF* running time versus input size in the largest dataset. As shown, when input size becomes larger, *transfer* time linearly increases while *SMF* growth is closer to linear order than $O((n+m)^2)$ which implies $\frac{\bar{w}}{W_a} = O(\frac{1}{n+m})$.

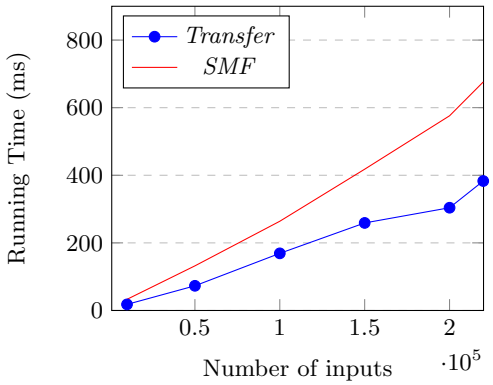

Figure 4: *Transfer* time and *SMF* running time for various input sizes. *SMF* is almost linear.

3.5 Common MBR Filter

CMF is an additional level of filtering that is applied on polygon edges to reduce number of candidate polygon pairs as well as the number of edges to be considered in the refinement phase by eliminating those edges that do not intersect the Common MBR. Given a pair $(P1, P2) \in \mathbf{C}$, with corresponding MBRs, MBR_{P1} and MBR_{P2}, their Common MBR ($MBR_{P1 \cap P2}$) is defined as the area covered by both of them (see green rectangles in Figure 5).

Algorithm 2 shows how *CMF* eliminates more polygon pairs from *SMF* output set **C** and classifies the remaining pairs into two groups for *point-in-polygon* and *edge-intersection* tests for the refinement phase while it eliminates all the non-intersecting edges from each polygon which does not intersect with the respective Common MBRs. The correctness of Algorithm 2 will be shown through Lemma 2 and 3.

LEMMA 2. ***CMF-Pre-PnP Test**: Given polygon pair $(P1, P2) \in \mathbf{C}$ with corresponding minimum bounding rectangles MBR_{P1} and MBR_{P2}, if $P1$ contains $P2$, then MBR_{P1} contains MBR_{P2}. In other words, $MBR_{P1 \cap P2} = MBR_{P2}$.*

PROOF. MBR_{P1} contains P_1 from definition. Also P_1 contains P_2 from Lemma assumption. Applying transitive property over contain relation leads to MBR_{P1} contains P_2.

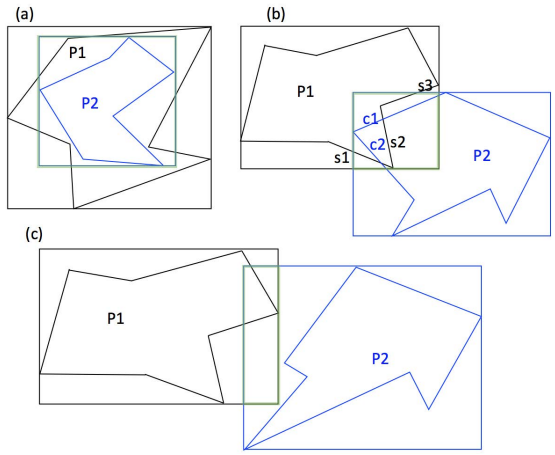

Figure 5: Examples for three *CMF* output classes: (a) $MBR_{P1\cap P2} = MBR_{P2}$ and CMF will tag $(P1, P2)$ for "P1 contains P2" *point-in-polygon* test. (b) $MBR_{P1\cap P2} \neq MBR_{P1}$ and $MBR_{P1\cap P2} \neq MBR_{P2}$ and $P1 \cap MBR_{P1\cap P2} \neq \emptyset$ and $P2 \cap MBR_{P1\cap P2} \neq \emptyset$, therefore (P_1, P_2) is directly sent to *edge-intersection* test. (c) $P2 \cap MBR_{P1\cap P2} = \emptyset$ which means the pair is disjoint.

Now lets assume $MBR_{P1\cap P2} = MBR_c \neq MBR_{P2}$. Because P_2 is inside of both MBR_{P1} and MBR_{P2}, it is also inside of MBR_c and since MBR_{P2} contains MBR_c and $MBR_c \neq MBR_{P2}$, MBR_c is minimum bounding rectangle of P_2 that is on the contrary with lemma assumption (minimum bounding rectangle of P_2 is MBR_{P2}). Thus, $MBR_{P1\cap P2} = MBR_{P2}$ □

Lemma 2 provides a necessary condition for *point-in-polygon* test. We can classify a given polygon pair of **C**, $(P1, P2)$, into one of the three following categories:1) Pairs with $MBR_{P1\cap P2} = MBR_{P1}$, 2) Pairs with $MBR_{P1\cap P2} = MBR_{P2}$, and 3) Pairs with partially-overlapping MBRs (see Figure 5). The first two classes can be added to *within candidate set* **W** for actual *point-in-polygon* test. By applying Lemma 2 before doing this test over all **C** elements, we gain performance due to the following reasons:

- For any given pair, verifying whether Lemma 2 holds true is only a constant-time operation while actual *point-in-polygon* test takes $O(n_e)$ where n_e is the number of edges.
- Join predicate requires testing for both "P1 contains P2" and "P2 contains P1" cases, but Lemma 2 identifies which polygon may contain the other one that eliminates one unnecessary test.

Later in Section 3.5.1, we provide more analysis.

LEMMA 3. ***CMF-Pre-Edge-intersection Test**: Given two edges $E_{P1}(i)$ and $E_{P2}(j)$ from polygons $P1$ and $P2$, if the edges intersect, then they either completely lie inside $MBR_{P1\cap P2}$ or intersect it. In either case, their intersection point is not outside $MBR_{P1\cap P2}$.*

PROOF. For the explanation refer to [6]. □

Algorithm 2 Common MBR filtering algorithm

Input: set **C** **Output**: sets **W** and **I**

```
1: procedure CMF-FILTER
2:   for each pair (i, j) ∈ C do
3:     if MBR_{i∩j} == MBR_i then
4:       W ← W ∪ (i, j) for polygon i inside polygon j test
5:     else if MBR_{i∩j} == MBR_j then
6:       W ← W ∪ (i, j) for polygon j inside polygon i test
7:     else
8:       Ê_i ← {E_i(k) | E_i(k) intersects MBR_{i∩j}}
9:       Ê_j ← {E_j(k) | E_j(k) intersects MBR_{i∩j}}
10:      if |Ê_i| == 0 or |Ê_j| == 0 then
11:        Discard (i,j)
12:      else
13:        I ← I ∪ (i, j)
14:      end if
15:    end if
16:  end for
17: end procedure
```

Lemma 3 provides a necessary condition for *edge-intersection* test. It says that given $(P1, P2)$ pair, if any edge from P1 lies completely outside of $MBR_{P1\cap P2}$, it will not intersect with P2. As a result, we can remove that edge from polygon edge list for the refinement phase. The following Corollary is a direct result of Lemma 2 and 3 and it can be used to detect some disjoint pairs in **C** before refinement phase.

COROLLARY 1. *Given pair $(P1, P2) \in \boldsymbol{C}$, let $\hat{E}_{P1} = \{i | E_{P1}(i)$ either intersects $MBR_{P1\cap P2}$ or lies inside it$\}$. Similarly, we can define $\hat{E}_{P2}$, intersecting-edge candidate set for $P2$. $P1$ and $P2$ are disjoint if $(P1, P2) \notin \boldsymbol{W}$, the within candidate set, and $\hat{E}_{P1} = \emptyset$ or $\hat{E}_{P2} = \emptyset$*

PROOF. It can be derived directly from Lemma 2 and Lemma 3. □

To illustrate Lemma 2 and 3 and Colorrary 1, three different examples are shown in Figure 5. Figure 5 (a) shows a case for Lemma 2 where $MBR_{P1\cap P2} = MBR_{P2}$. In this example, *CMF* assigns $(P1, P2)$ to *within candidate set* for point-in-polygon test for "P1 contains P2" case. In Figure 5 (b), $MBR_{P1\cap P2}$ is equal to none of the MBRs. As such, using Lemma 3, *CMF* makes *intersecting-edge candidate sets* for $P1$ and $P2$ that are $\hat{E}_{P1} = \{s1, s2, s3\}$ and $\hat{E}_{P2} = \{c1, c2\}$ respectively. In this case, because neither of $\hat{E}_{P1}$ and $\hat{E}_{P2}$ is empty, *CMF* categorizes $(P1, P2)$ into *Intersecting-Edge candidate set*. Later, *EI_Test* will use only $\hat{E}_{P1}$ and $\hat{E}_{P2}$ instead of E_{P1} and E_{P2} for refinement phase. Figure 5 (c) is an example of Corollary 1 as $(P1, P2) \notin \mathbf{W}$ and $\hat{E}_{P2} = \emptyset$. Therefore, *CMF* identifies this pair as disjoint and just discards it.

CMF thus classifies elements of **C** as follows:

1. *Within candidate set* (**W**): set of all the polygon pairs $(P1, P2) \in \mathbf{C}$ such that $MBR_{P1\cap P2}$ is either equal to MBR_{P1} or MBR_{P2}.
2. *Intersecting-edge candidate set* (**I**): set of all polygon pairs $(P1, P2) \in \mathbf{C}$ such that $(P1, P2) \notin \mathbf{W}$ and $\hat{E}_{P1}$ and $\hat{E}_{P2}$ are non-empty.

3. *Disjoint set*: Polygon pairs $(P1, P2) \in \mathbf{C}$ that are neither in $\mathbf{W}$ nor in $\mathbf{I}$.

As *CMF* iterates through each edge once, the algorithm complexity is $O(n_e)$ which n_e is number of edges and its implementation is straightforward. For a polygon pair $(P1, P2)$, we assign a two dimensional GPU block to handle edges of each polygon in a separate block dimension. Polygon edges are evenly distributed among threads in each dimension to make the algorithm load-balanced. Each thread verifies Lemma conditions for its data and partially keeps list of potentially intersecting edges. Finally, using shared memory and reduction tree, algorithm classifies pairs in $\mathbf{C}$ based on results and prepares *intersecting-edge candidate sets* $\hat{E}_{P1}$ and $\hat{E}_{P2}$.

3.5.1 CMF Analysis

For a given candidate set $\mathbf{C}$, we define *edge-reduction factor* as

$$R_E = \frac{\sum_{(i,j)\in\mathbf{C}} |E_i| + |E_j|}{\sum_{(i,j)\in\mathbf{C}} |\hat{E}_i| + |\hat{E}_j|} \tag{4}$$

In the worst-case, all edges may lie inside their *common MBR* or intersect it and $R_E = 1$, but based on our experimental results, shown in Table 3, $R_E \approx 40$ which shows effectiveness of *CMF* in pruning polygons before refinement phase. Table 3 shows timing and workload of *edge-intersection* test with and without *CMF*. *CMF* eliminates almost two-third of pairs by applying Lemma 3. It also, makes *edge-intersection* test almost 30 times faster by making *intersecting-edge candidate set* 40 times smaller than all the edges.

Table 3: CMF effect on reducing workload of the refinement phase for Water datasets.

	No CMF	*With CMF*
Time (ms)	120,751	4,401
♯ of Edge-intersecting pairs	566,656	198,142
♯ of edges (layer1)	1,048,479,573	25,969,322
♯ of edges (layer2)	954,431,290	20,451,866

The following lemma proves that even in the worst-case scenario, *CMF* will reduce number of operations in our point-in-polygon test.

LEMMA 4. ***PnP analysis**: Given a candidate set $\mathbf{C}$ of potentially intersecting polygons, applying CMF filter will always reduce the overall work for point-in-polygon test.*

PROOF. Equation 5.a and 5.b represent the number of operations in *point-in-polygon* test with and without *CMF* filter.

$$\begin{aligned} & k \cdot |\mathbf{C}| + \bar{N}_e \cdot |\mathbf{W}| \qquad (a) \\ & 2 \cdot \bar{N}_e \cdot |\mathbf{C}| \qquad (b) \end{aligned} \tag{5}$$

where $\bar{N}_e$ is the average number of edges in polygons and k is a constant factor such that in general, $k < \bar{N}_e$ ($k \approx 8$ as we only need to test for two MBR equalities each with 4 coordinate values). We need to show $5.a < 5.b$. Since $0 \leq |\mathbf{W}| \leq |\mathbf{C}|$ and $k < \bar{N}_e$, we have $\bar{N}_e \cdot |\mathbf{W}| < (2 \cdot \bar{N}_e - k) \cdot |\mathbf{C}|$. As such the condition always holds true. □

Figure 6 shows *CMF* running time for various data sizes. As shown, *CMF* running time increases linearly as data size becomes larger.

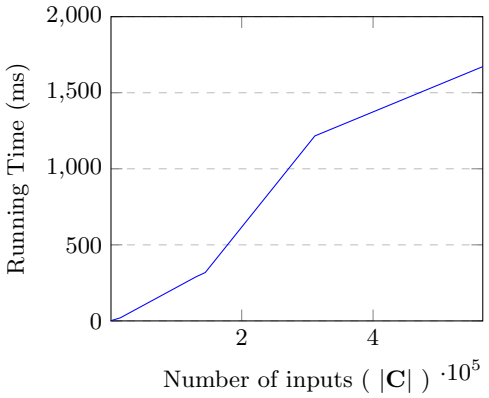

Figure 6: Running time of *CMF* versus various range of input sizes.

3.6 Refinement Algorithms

The refinement phase removes all the non-intersecting polygon pairs by finding the pairs with intersecting edges or detecting if one of the polygons lies inside the other one. Sequential plane sweep-based edge-intersection algorithms are generally used in the refinement phase but these methods have not been proven suitable for fine-grained data parallel processing over GPUs [3]. While some work has been done to parallelize the plane sweep on CPU [10], none of the proposed candidates result in an algorithm amendable to fine-grained SIMD parallelism such as with GPUs. Our approach is embarrassingly parallel. For a given polygon pair (i, j), it performs an all-to-all *edge-intersection* test with $O(|\hat{E}_i| \cdot |\hat{E}_j|)$ time complexity.

As we described in Section 3.3, our refinement phase includes two subsystems. Although we have not developed any new algorithm for these two components, efficiently parallelizing their sequential counterparts over GPU is not a trivial task and requires some design changes to fit them into shared memory model. In the following subsections, we explain some of their implementation details.

3.6.1 Parallel Point-in-Polygon Test

To detect if a polygon is inside the other polygon, we have used *point-in-polygon* test. We apply crossing test method to detect if a given test point is inside a polygon [17]. The technique is to shoot a ray from the test point along an axis and count number of crossings of the polygon edges to check if it is odd. Sequential implementation of this algorithm uses a *for loop* that iterates through all vertices of a given polygon. We have implemented this algorithm over GPU by breaking down this *for loop* by distributing equitably among different threads of a GPU-block. Finally, using a reduction tree algorithm, we combine the partial results from different threads.

We compare load-balanced PnPTest with two other algorithms 1) sequential version of crossing test and 2) naive *point-in-polygon* test over GPU in which each thread is responsible for the entire *for loop* and there is no workload distribution for a given test point. The results are summarized in Table 4. *PnPTest* speedup is in the 28 to 30-fold range compared to the sequential version. Our load-balanced *point-in-polygon* test also achieved a good speedup

in 8 to 9-fold range versus naive GPU version which demonstrates the importance of GPU-load-balancing.

Table 4: Running time of *PnPTest* versus sequential

Dataset	$\hat{N}_e.\|\mathbf{W}\|$	Running time (ms)		
		Seq.	GPU	
			Naive	PnPTest
Urban	11,266,110	672	210	24
Telecom	7,615,041	165	45	6
Water	105,314,500	22,058	6,459	725

PnPTest running time of *Urban* dataset is greater than *Telecom* dataset while number of polygons in *Telecom* is much larger because as shown in Lemma 4, the time complexity of our *point-in-polygon* algorithm is $O(\hat{N}_e.|\mathbf{W}|)$.

3.6.2 Parallel Edge-Intersection Test

As we explained in Section 3.3, elements of $\mathbf{I}$ and those pairs from $\mathbf{W}$ which do not pass *point-in-polygon* test successfully along with *intersecting-edge candidate sets* generated by *CMF* are sent to *edge-intersection* test component. We implemented a load-balanced *edge-intersection* algorithm using a shared memory model. The algorithm assigns a GPU-block to each pair of polygons. Within a block, edge pairs are distributed evenly among threads. For actual intersection test, the algorithm calculates intersection point of a given edge pair, then tests if this point lies on the both edge segments (and not outside them). We have implemented this algorithm efficiently by applying two optimization techniques:

- Calculating line intersection requires floating point computations which is the most time consuming operation in any processor. To optimize the algorithm, first we test if MBRs built from edges have overlap and then, calculate intersection point for only the MBR-overlapping edge pairs.

- All threads within a block work over the same polygon pair and once a thread finds two cross-layer intersecting-edges, the polygon pair can be identified as output. As a result, if a thread detects such a case, it sends signal to other threads in the block to terminate. This makes their resources available for other blocks.

3.6.3 Refinement Analysis

Figure 7 shows running time for *PnPTest* and *EITest* for various input sizes.

As shown, *PnPTest* has a linear running time while *EITest* is $O(n^2)$. As *point-in-polygon* test iterates through all vertices of a given polygon, linear complexity is expected. On the other hand, the complexity of *EITest* is quadratic, because *edge-intersection* algorithm, for a given pair $(P1, P2)$ with *intersecting-edge candidate sets* $\hat{E}_{P1}$ and $\hat{E}_{P1}$, tests for all to all cross-layer edge-intersection. As a result, complexity is $O(|\hat{E}_{P1}| \cdot |\hat{E}_{P2}|)$.

4 Performance Evaluation

First, we describe the experimental setup and then we compare the results with other base-line methods.

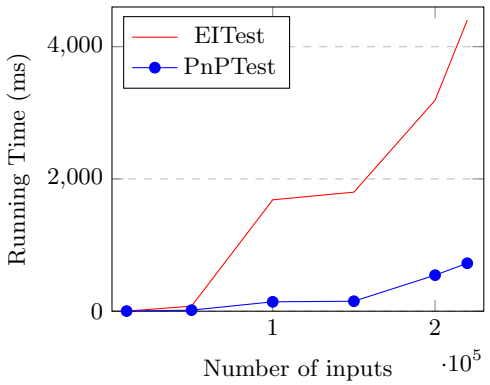

Figure 7: Linear running time of *PnPTest* versus quadratic complexity of *EITest* for different input sizes.

4.1 Experimental Setup

We have done all the GPU experiments on a compute node that has 12-core Intel Xeon CPUE5-2650 CPU running at a clock speed of 2.0 GHz with 64GB of main memory. The node is equipped with a NVIDIA GTX 780 GPU that has 6GB of memory with 288.4 GB/sec memory bandwidth. We have verified that similar results are obtained on other NVIDIA GPU's including *Tesla K40* models.

To the best of our knowledge, there is no GPU-based work which has implemented *ST_Intersect* operation. Therefore, we used PostgreSQL version 9.4 with PostGIS version 2.2 and *GEOS* library version 3.4.2 [12] as sequential baselines for comparison with *GCMF*. PostGIS is a spatial database extender for the PostgreSQL object-relational database. It adds support for geographic objects allowing spatial queries to be run in SQL. We ran PostGIS on a desktop with 3.6 GHz processor with 16 GB of main memory. Since our datasets are shapefiles, we use *shp2pgsql* tool in *PostGIS* for converting shape files into database tables. *ST_Intersect* predicate was used in spatial join query [12]. We also used *Intersects* method of *PreparedGeometry* class that is the optimized and indexed implementation of Geometry class of *GEOS* $C++$ library. *GEOS* experiments are done on a node equipped with *2.6 GHz Intel Xeon E5-2660v3* processor in the *Roger NCSA* cluster.

4.2 Results

Table 5 shows end-to-end running time of different algorithms on different datasets including CPU-GPU transfer times. The relative speedup gain for our GPU-based system is up to 39-fold versus both *GEOS* library and *PostGIS* software.

Table 5: End-to-end running time for three different methods

Dataset	Running time (ms)			♯ of Outputs
	PostGIS	GEOS	GCMF	
Urban	3,120	5,770	149	23,634
Telecom	17,900	8,200	560	581,351
Water	232,122	148,040	7,856	539,974

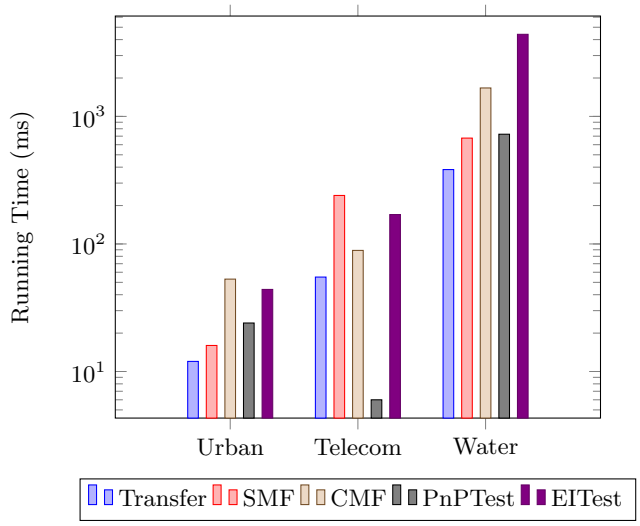

Figure 8: Detailed running timing of system components for Urban, Telecom and Water datasets (log scale)

4.2.1 System Component Analysis

We have provided detailed running time of each system component in Table 6 and Figure 8 for all datasets. As value ranges are large, time in the bar chart is presented in the logarithmic scale for easier comparison.

Although by applying *CMF* filter, we tried to reduce *edge-intersection* processing time, this component still takes more than half of the total running time.

Table 6: Detailed running time of system components for all three datasets

Dataset	Detailed running time (ms)				
	Transfer	*SMF*	*CMF*	*PnPTest*	*EITest*
Urban	12	16	53	24	44
Telecom	55	240	89	6	170
Water	383	676	1671	725	4401

Our real datasets are heterogeneous in size. They include polygons with various ranges, from less than 100 vertices up to 50,000 vertices, that makes load-balancing task hard. We tried evenly distributing the *edge-intersection* test over all threads across all blocks by assigning a constant number of tests to each thread. But because of inefficient use of memory bandwidth and other GPU resources, the performance was worse than our simpler current method. Determining the impact of this load-imbalance on the performance requires comprehensive understanding of the behavior of GPUs under these circumstances that is part of our future work.

Finally, Table 7 summarizes time and space complexities of each phase of *GCMF* system. In the filtering phases, *GCMF* estimates the required memory space for the output and then allocates the memory. To estimate space, we have used two strategies. 1) Roughly estimating the number of outputs in *SMF*. 2) Counting the exact number of outputs and then allocate the memory in *CMF*. Therefore, for these components, the space complexity is proportional to their corresponding outputs k_1 and k_2, respectively. Our space-optimized design lets the system handle large datasets on a single GPU. Also, time complexity of most of the phases is linear. *Edge-intersection* is an exception, however, because of effective linear filters (*SMF* and *CMF*) applied before it, workload of this operation significantly reduces. As Table 7 shows, for *Water* dataset, more than 99% of all possible polygon pairs are eliminated by *SMF*, and then *CMF* further reduces 65% of the remaining pairs. Finally, the total running time of the filter phases is less than one-third of the overall execution time, which shows the effectiveness of the proposed filtering methods. However, *Edge-intersection* test is still the most time-consuming operation as it takes more than 50% of the execution time.

5 Conclusion

In this paper, we have introduced *GCMF*, an end-to-end spatial join system for non-indexed polygonal data over a single GPU platform. The system included 4 subsystems: two filtering components as well as *point-in-polygon* test and *edge-intersection* test subsystems. We proposed sort-based MBR filtering algorithm for GPU with linear average time complexity. Also, we introduced *CMF* with linear time complexity as an efficient filtering technique to reduce the number of polygon candidate pairs before the refinement phase. We also have shown that *CMF* reduces the size of remaining candidate pairs by pruning disjoint edges apriori. Our experimental results over real datasets yielded up to 39-fold relative speedup gain versus optimized sequential *GEOS* library and *Postgres* with *PostGIS* spatial database system. Moreover, it confirmed the efficiency of *CMF* in removing about two-third of pairs from the set of candidate polygons before *edge-intersection*. It also reduced the size of polygon pairs for refinement phase up to 40-fold smaller.

Our plan is to integrate this system into a MPI based system which can partition 1-3 order larger datasets among the compute nodes, such as our MPI-GIS system [15] [16] that has a potential for speeding up such systems by 1-2 orders of magnitude by effectively employing GPUs.

6 References

[1] Aji, A., Teodoro, G., and Wang, F. Haggis: Turbocharge a mapreduce based spatial data warehousing system with GPU engine. In *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (2014), ACM, pp. 15–20.

[2] Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., and Vitter, J. S. Scalable sweeping-based spatial join. In *VLDB* (1998), vol. 98, Citeseer, pp. 570–581.

[3] Audet, S., Albertsson, C., Murase, M., and Asahara, A. Robust and efficient polygon overlay on parallel stream processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2013), ACM, pp. 304–313.

[4] civilmap. Available at: https://blog.civilmaps.com/the-lidar-data-crunch/.

[5] García, Y. J., Lopez, M. A., and Leutenegger, S. T. On optimal node splitting for R-trees. In

Table 7: Running Time and Workload Reduction of Each Component for Water dataset

	SMF	CMF	PnP	EI
Space Complexity	$O(k_1)$	$O(k_2)$	$O(n)$	$O(n)$
Time Complexity	$O(n \cdot b + k_1)$	$O(n)$	$O(n)$	$O(n^2)$
n: # of inputs (Polygon pairs)	$219,831 \times 463,591$	1,020,458	842,516	198,142
k: # of outputs (Polygon pairs)	1,020,458	198,142	453,802	86,172
Reduction ($\frac{k}{n}$)	$> 99\%$	65%	46%	57%
Time Fraction	8.6%	21.27%	9.23%	56.02%

Proceedings of the 24rd International Conference on Very Large Data Bases (1998), Morgan Kaufmann Publishers Inc., pp. 334–344.

[6] Jacox, E. H., and Samet, H. Spatial join techniques. *ACM Transactions on Database Systems (TODS) 32*, 1 (2007), 7.

[7] Leutenegger, S. T., Lopez, M., Edgington, J., et al. Str: A simple and efficient algorithm for R-tree packing. In *Data Engineering, 1997. Proceedings. 13th International Conference on* (1997), IEEE, pp. 497–506.

[8] Lieberman, M. D., Sankaranarayanan, J., and Samet, H. A fast similarity join algorithm using graphics processing units. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on* (2008), IEEE, pp. 1111–1120.

[9] Luo, L., Wong, M. D., and Leong, L. Parallel implementation of R-trees on the GPU. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific* (2012), IEEE, pp. 353–358.

[10] McKenney, M., and McGuire, T. A parallel plane sweep algorithm for multi-core systems. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2009), ACM, pp. 392–395.

[11] Pavlovic, M., Tauheed, F., Heinis, T., and Ailamakit, A. GIPSY: joining spatial datasets with contrasting density. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (2013), ACM, p. 11.

[12] PostGIS. http://postgis.net/.

[13] Prasad, S. K., McDermott, M., He, X., and Puri, S. GPU-based parallel R-tree construction and querying. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International* (2015), IEEE, pp. 618–627.

[14] Prasad, S. K., McDermott, M., Puri, S., Shah, D., Aghajarian, D., Shekhar, S., and Zhou, X. A vision for GPU-accelerated parallel computation on geo-spatial datasets. *SIGSPATIAL Special 6*, 3 (2015), 19–26.

[15] Puri, S., and Prasad, S. K. MPI-GIS: New parallel overlay algorithm and system prototype.

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

[17] Shimrat, M. Algorithm 112: position of point relative to polygon. *Communications of the ACM 5*, 8 (1962), 434.

[18] Simion, B., Ray, S., and Brown, A. D. Speeding up spatial database query execution using gpus. *Procedia Computer Science 9* (2012), 1870–1879.

[19] Stonebraker, M., Frew, J., Gardels, K., and Meredith, J. The sequoia 2000 storage benchmark. In *ACM SIGMOD Record* (1993), vol. 22, ACM, pp. 2–11.

[20] Yampaka, T., and Chongstitvatana, P. Spatial join with R-tree on graphics processing units. *KMUTNB: International Journal of Applied Science and Technology 5*, 3 (2013), 1–7.

[21] You, S., Zhang, J., and Gruenwald, L. Parallel spatial query processing on gpus using R-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (2013), ACM, pp. 23–31.

[22] You, S., Zhang, J., and Gruenwald, L. Scalable and efficient spatial data management on multi-core CPU and GPU clusters: A preliminary implementation based on impala. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on* (2015), IEEE, pp. 143–148.

[23] Zhang, J., and You, S. CudaGIS: report on the design and realization of a massive data parallel GIS on GPUs. In *Proceedings of the Third ACM SIGSPATIAL International Workshop on GeoStreaming* (2012), ACM, pp. 101–108.

[24] Zhang, J., and You, S. 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* (2012), ACM, pp. 23–32.

[25] Zhang, J., You, S., and Gruenwald, L. High-performance spatial query processing on big taxi trip data using gpgpus. In *Big Data (BigData Congress), 2014 IEEE International Congress on* (2014), IEEE, pp. 72–79.

[26] Zhou, X., Abel, D. J., and Truffet, D. Data partitioning for parallel spatial join processing. *Geoinformatica 2*, 2 (1998), 175–204.