Marquette University
e-Publications@Marquette

Dissertations (1934 -) Dissertations, Theses, and Professional Projects

Hierarchical and Adaptive Filter and Refinement Algorithms for Geometric Intersection Computations on GPU

Yiming Liu
Marquette University

Follow this and additional works at: https://epublications.marquette.edu/dissertations_mu

Part of the Computer Sciences Commons

Recommended Citation

Liu, Yiming, "Hierarchical and Adaptive Filter and Refinement Algorithms for Geometric Intersection Computations on GPU" (2021). *Dissertations (1934 -)*. 1064.
https://epublications.marquette.edu/dissertations_mu/1064

HIERARCHICAL AND ADAPTIVE FILTER AND REFINEMENT ALGORITHMS FOR GEOMETRIC INTERSECTION COMPUTATIONS ON GPU

by

Yiming Liu, B.Eng., M.S.

A Dissertation submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy

Milwaukee, Wisconsin

May 2021

ABSTRACT
HIERARCHICAL AND ADAPTIVE FILTER AND REFINEMENT ALGORITHMS FOR GEOMETRIC INTERSECTION COMPUTATIONS ON GPU

Yiming Liu

Department of Computer Science
Marquette University, 2021

Geometric intersection algorithms are fundamental in spatial analysis in Geographic Information System (GIS). This dissertation explores high performance computing solution for geometric intersection on a huge amount of spatial data using Graphics Processing Unit (GPU). We have developed a hierarchical filter and refinement system for parallel geometric intersection operations involving large polygons and polylines by extending the classical filter and refine algorithm using efficient filters that leverage GPU computing.

The inputs are two layers of large polygonal datasets and the computations are spatial intersection on pairs of cross-layer polygons. These intersections are the compute-intensive spatial data analytic kernels in spatial join and map overlay operations in spatial databases and GIS. Efficient filters, such as PolySketch, PolySketch++ and Point-in-polygon filters have been developed to reduce refinement workload on GPUs. We also showed the application of such filters in speeding-up line segment intersections and point-in-polygon tests. Programming models like CUDA and OpenACC have been used to implement the different versions of the Hierarchical Filter and Refine (HiFiRe) system.

Experimental results show good performance of our filter and refinement algorithms. Compared to standard R-tree filter, on average, our filter technique can still discard 76% of polygon pairs which do not have segment intersection points. PolySketch filter reduces on average 99.77% of the workload of finding line segment intersections. Compared to existing Common Minimum Bounding Rectangle (CMBR) filter that is applied on each cross-layer candidate pair, the workload after using PolySketch-based CMBR filter is on average 98% smaller. The execution time of our HiFiRe system on two shapefiles, namely USA Water Bodies (contains 464K polygons) and USA Block Group Boundaries (contains 220K polygons), is about 3.38 seconds using NVidia Titan V GPU.

ACKNOWLEDGEMENTS

Yiming Liu

I would like to express my gratitude to my advisor Dr. Satish Puri for his invaluable guidance and encouragement throughout my graduate study. His insightful feedback brought my work to a higher level. I would also like to thank the other members of my thesis committee, Dr. Praveen Madiraju and Dr. Debbie Perouli for their valuable time and great comments. In addition, I would like to thank all members of the parallel computing lab for the support and advice.

I would like to express my deep appreciation to my parents who always love and support me. I would also like to thank my friends who always encourage me.

TABLE OF CONTENTS

ACKNOWLEDGEMENT . . . . **i**

LIST OF TABLES . . . . **vi**

LIST OF FIGURES . . . . **viii**

1 INTRODUCTION **1**

1.1 Introduction 1

1.2 Motivation 2

2 LITERATURE REVIEW **5**

2.1 Introduction 5

2.2 Minimum Bounding Box and Related Algorithms 7

2.2.1 Clipped Bounding Box 7

2.2.2 Common Minimum Bounding Rectangle 9

2.3 Polygon Intersection Related Algorithms 10

2.3.1 Traditional Algorithms for Polygon Intersection 10

2.3.2 GPU-based Algorithms for Spatial Aggregation 12

2.4 Spatial Analytics System and Hybrid Platform 16

2.4.1 CPU-GPU Hybrid Platform 16

2.4.2 Parallel Implementation on CPU-GPU 18

2.4.3 Modern Spatial Analytics Systems 19

3 POLYSKETCH: AN EFFICIENT FILTER FOR GEOMETRIC INTERSECTION COMPUTATIONS USING GPU **22**

3.1 Introduction 22

3.2 Related Work 23

3.3 Geometric Intersection Computations 24
3.3.1 Problem Description 24
3.3.2 PolySketch Algorithm 25
3.4 Experimental Setup and Results 28
3.4.1 Datasets 28
3.4.2 Hardware Description 30
3.4.3 Performance of PolySketch Compared to CMBR 30
3.4.4 Performance of Using Different PolySketch Sizes 36
3.5 Conclusion 37
4 HIERARCHICAL FILTER AND REFINEMENT SYSTEM OVER LARGE POLYGONAL DATASETS ON CPU-GPU **38**
4.1 Introduction 38
4.2 Motivation 39
4.3 Hierarchical Filter and Refinement System 40
4.3.1 System Design Overview 40
4.3.2 CMBR Filter Based Task Reduction 42
4.3.3 Workload Reduction by PolySketch Filter 43
4.3.4 PNP Based Task Reduction Algorithm 43
4.3.5 Striping Algorithm and Stripe-based PNP Function . . . 47
4.4 Performance Evaluation 48
4.4.1 Performance of PNP Filters 48
4.4.2 Execution Time Results 51
4.4.3 CUDA information 52

4.5 Conclusion 53

5 EFFICIENT FILTERS FOR GEOMETRIC INTERSECTION COMPUTATIONS USING GPU **54**

5.1 Introduction 54

5.2 Related Work 55

5.3 PSCMBR Algorithm 58

5.3.1 Overview of PSCMBR Filter 58

5.3.2 Advantages of PSCMBR Filter 62

5.3.3 The Implementation of PSCMBR Filter 63

5.3.4 Optimization: Mapping Tiles to Threads 66

5.4 Performance Evaluation 66

5.4.1 PSCMBR Filter Performance 66

5.4.2 Performance of Using Different PSCMBR Tile-sizes . . . 69

5.5 Point-in-Polygon Filter Using PolySketch 70

5.5.1 Algorithm Overview 72

5.5.2 Comparison of PNP Workload Reduction 73

5.5.3 The Implementation of PNP Filters 75

5.6 Performance Evaluation 75

5.6.1 System Performance with PNP Filters 75

5.6.2 Filters with PNP Test Workload 76

5.6.3 New Hierarchical Filter and Refine System 78

5.7 Conclusion 79

6 ADAPTIVE FILTER FOR GEOMETRIC INTERSECTION WITH RECTANGLE INTERSECTION COMPUTATIONS USING GPU **80**

6.1 Introduction 80

6.2 Related work 81

6.3 PolySketch++ Algorithm 82

6.3.1 Overview of Adaptive PolySketch 82

6.3.2 Advantages of Adaptive PolySketch 87

6.3.3 The Implementation of Adaptive PolySketch and CUDA Dynamic Parallelism 88

6.4 Experimental Setup and Results 90

6.4.1 Data Sets 90

6.4.2 Adaptive PolySketch Run-Time 91

6.4.3 Effect of Varying Tile Sizes and Threads 92

6.4.4 Adaptive PolySketch Workload 94

6.4.5 Execution Time Comparison Using Different GPUs . . . 96

6.5 Rectangle Intersection Filter on GPU 96

6.5.1 Overview of LMBR Filter 96

6.5.2 The Implementation of LMBR Filter 98

6.6 LMBR Filter Performance 100

6.7 Conclusion 101

7 CONCLUSION AND FUTURE WORK **102**

7.1 Conclusion 102

7.2 Future Work 102

BIBLIOGRAPHY **104**

LIST OF TABLES

3.1 Three real datasets used in our experiments 29

3.2 Sketch and CMBR effect on the LSI function for Water dataset . . . 31

3.3 Sketch and CMBR effect on the LSI function for Urban dataset . . . 32

3.4 Sketch and CMBR effect on the LSI function for Lakes dataset 32

3.5 Sketch and CMBR effect on reducing tasks of the LSI function for different datasets 33

3.6 Sketch effect on reducing workload and line segments by the LSI function for three datasets using percentage. 34

3.7 CMBR effect on reducing workload and line segments by the LSI function for three datasets using percentage. 34

3.8 The performance of using different tile-size for Water dataset 36

4.1 Run-times by using different GPUs without filters 39

4.2 Candidate pairs before hierarchical filtering 41

4.3 Eliminating line segments by hierarchical filtering 41

4.4 PNP based task reduction algorithm and striping effect on reducing workload of the PNP function for both the datasets and the reduction percentage 48

4.5 Striping effect on reducing workload for the Stripe-based PNP function for both the datasets 50

4.6 PNP based Task Reduction Algorithm effect on reducing workload when a polygon MBR is inside another polygon MBR 50

4.7 Execution time in seconds 51

4.8 Execution time breakdown details for Water dataset. (NA means it is not applicable.) 52

4.9 Execution time breakdown details for Urban dataset. (NA means it is not applicable.) 52

5.1 Symbol Table 60

5.2 An example of input polygon pair in Figure 5.4 62

5.3 Effect of different filters on the LSI function for Water dataset 67

5.4 Effect of different filters on the LSI function for Urban dataset 67

5.5 Effect of different filters on the LSI function for Lake dataset 68

5.6 Performance variation while using different tile-sizes for Water dataset 70

5.7 System run-time (does not include R-tree time) 75

5.8 Workload using different methods in tasks where two polygons have line segment intersections. 76

5.9 The workload in PNP test for the tasks where one polygon may be totally inside another polygon 78

6.1 Symbol Table 85

6.2 Three real data sets used in our experiments for PolySketch++ . . . 90

6.3 The run-time using static and adaptive tile size. 91

6.4 The LSI function workload and tile workload 94

6.5 The run-time of PolySketch (P) and PolySketch++ (P++) using large polygons (around 25000 vertices) 95

6.6 The run-time of PolySketch and PolySketch++ with LSI function using Titan V and Titan Xp 96

6.7 The execution time of using R-tree, all-to-all method or LMBR filter 100

6.8 Time performance of LMBR by partitioning both layers and using different LMBR sizes 100

6.9 Time performance of LMBR by partitioning 1 layer and using different LMBR sizes 100

LIST OF FIGURES

2.1 Filter and refine technique for one polygon pair 5

2.2 Common minimum bounding rectangle examples (Red rectangle is CMBR.) 10

2.3 An example of raster join approach; (a) input data, (b) all points are rendered onto an FBO, (c) the pixel values are aggregated corresponding to fragments of a polygon [1] 13

2.4 Accurate raster join: (a) forming boundaries of polygons, and (b) 'draw' polygons [1] 15

3.1 (a) Line segment intersection vertices, (b) Vertices inside another polygon, and (c) Output polygon (Best Viewed in Color) 24

3.2 PolySketch of polyline data composed of tiling at three different levels. 26

3.3 Improve approximation using PolySketch 27

3.4 Polygon intersection using PolySketch. Tiles for each polygon is shown in different colors. 27

3.5 The workload in LSI function after using CMBR or Sketch. 35

4.1 Classifying PNP tasks after CMBR Filter 44

4.2 PNP cases: (a) Two polygons have line segment intersections, (b) and (c) Two polygons *touch* each other, (d) Two polygons' MBRs overlap but there is no actual intersection, (e) One polygon is inside another polygon but there is no line segment intersection, (f) The smaller polygon is not inside another polygon but the smaller MBR is inside another MBR. 45

4.3 An example of striping for Stripe-based PNP function 47

4.4 The percentage of different types of tasks after using PNP based task reduction algorithm 49

5.1 Two input polygons with (a) CMBR (green rectangle), (b) PolySketch showing the tiles and (c) only overlapping tiles after applying PolySketch filter. 56

5.2 PSCMBR filter with four tile-CMBRs. Only the common area of overlap between the candidate tile pairs are retained (see Figure 5.1(c)). . 59

5.3 PSCMBR filter for reporting line segment intersections (LSI). Two polygons A1 and B1 are the input and the output is list of intersections. 60

5.4 Effect of tile-size on the number of candidate tile-pairs. Input Tile-pairs%=$\frac{Tp*Tq}{P*Q}$*100. Candidate tile-pairs% = $\frac{C}{Tp*Tq}$*100. Each line denotes the output candidate tile-pairs% for a given input polygon-pair. 61

5.5 CUDA Implementation of PSCMBR Filter 65

5.6 An example of *intersection tile* (red rectangle) and *no-intersection tile* (yellow rectangle) 71

5.7 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. 74

5.8 Percentage of *Intersection tile* and *no intersection tile* for tasks where two polygons have line segment intersections 77

6.1 PolySketch++ filter example (a) Overlapping tiles highlighted by a circle. It shows a red tile overlapping with two green tiles, and (b) The overlapping tiles are further subdivided. Further refinement is not required. 82

6.2 Static filter shown in (a). Two-step adaptive filter shown in (b) and (c). (b) shows filter using coarse-grained tile size and (c) shows fine-grained tile size for overlapping tiles only; (d) An example where two polygons have large difference in sizes. 84

6.3 PolySketch++ filter for speeding up line segment intersections (LSI). The filter is dynamic and recursive. The Divide component produces more tiles (by factor of a and b) by subdivision of input tiles. 86

6.4 CUDA Implementation of PolySketch++ Filter 89

6.5 Execution-time variation using different parent tile-sizes (128 to 3072). Number of threads are adjusted based on tile-sizes (maximum number of threads are assigned to parent and child kernels based on the workload). 92

6.6 Effect of using different parent tile sizes and different number of threads assigned to the parent and child kernels 93

6.7 An example of LMBR 97

6.8 CUDA Implementation of LMBR Filter 99

Chapter 1
Introduction

1.1 Introduction

We are living in the era of Spatial Big Data. Spatial data is pervasive and growing rapidly. They are coming from billions of phones, cars, satellites, webs, and many different sources [2]. Location-based services are also increasing, such as Uber, Lyft, Twitter, location-tagged posts on Instagram, and so on [3]. In addition, with the development of data acquisition techniques, GPS-enabled devices, and their corresponding applications, spatial data are increasing at an unprecedented scale. Efficiently querying and analyzing such a huge volume of spatial data is challenging [4]. Spatial proximity queries between objects, spatial cross-matching queries, nearest neighbor queries, and window-based queries are general geospatial analyses. Combinations of these queries can be used in many cases, such as traffic analysis, travel patterns from massive GPS collections in transportation, post-processing large-scale climate model outputs, and so on [5]. Geometric intersection algorithms are also fundamental in spatial analysis [6, 7]. Computing the polygon intersection area [8, 9] is also significant to the fields of computer graphics, image processing, computer vision, and Geographic Information Systems (GIS). Since spatial data are increasing at an unprecedented scale, it is necessary to apply high-performance computing to spatial analysis.

In Geographic Information Systems (GIS) and spatial databases, vector geometries like polygons and polylines are used to represent real-world objects. The input to map overlay and spatial join queries are two layers of geo-spatial data. Spatial query operations often require expensive computational geometry algorithms. For computational efficiency, query operations are carried out in two phases. The first phase is a filter phase where complex geometries are approximated

by their minimum bounding rectangle (MBR). The filter phase employs spatial data structures like R-tree built using MBRs of geometries. Working with MBR representation is faster compared to actual geometries that may contain thousands of vertices. Geometries that could not possibly satisfy the query condition are removed. The output of the filter phase consists of candidate pairs that may or may not be part of the final output. The drawback of filtering using MBR is that it produces many false hits because of MBR approximation. As such, in the second phase of refinement, actual geometries are used to produce correct results by detecting and removing false hits.

Line Segment Intersection (LSI) and Point-in-Polygon (PNP) tests are invoked by geometric intersection algorithms. Given two layers R and S as inputs, spatial join returns the set of all pairs (r, s) where $r \in R$, $s \in S$, and their intersect predicates (such as, overlap, contain). Intersect predicate returns true if one polygon is inside another polygon or two polygons have line segment intersection(s). An example of a spatial join query is "Find all the places where roads cross rivers". Overlap is one of the spatial relations between r and s. Other spatial relations are *Intersects*, *Touches*, *Contains*, etc. Spatial databases and GIS support such topological relations on two layers of geometries. The output polygon produced by polygon intersection is not required. Polygon overlay should also calculate the output polygons, which requires finding all line segment intersections (by LSI test) and the vertices of a polygon that are inside another polygon (by PNP test). Compare to spatial join, the computation of polygon overlay is more complex and expensive.

1.2 Motivation

Spatial data is generated from a lot of real-world applications currently and the volume of them becomes much larger [2]. Efficiently querying and analyzing such a

huge volume of spatial data is time-consuming. Geometric intersection algorithms (such as spatial join and polygon overlay) are essential in spatial analysis in Geographic Information System. The goal of this dissertation is to study the current challenges of algorithms used in the filter and refine technique of geometric intersection computation and develop algorithms (mainly on GPU) to address some of them.

We also explore the filter and refine technique based on the CPU-GPU platform, which is robust to a variety of inputs. For example, the new filters will speedup "Jaccard Similarity" computation on GPU, which is used in solar flare prediction where polygonal data extracted from NASA's Solar Dynamics Observatory are used for polygon intersection and union to detect significant solar events [10]. In the biomedical domain, geometric intersection is used to analyze spatial features in digital pathology images containing millions of cells and terabytes in size. In both use cases, Jaccard similarity computation using such filters will make the cross-comparison computations fast and near-realtime. For disease diagnosis, cross-comparison of millions of such polygons will be undertaken. There are also spatial query cases in analytical medical imaging. COVID-19 has become a pandemic and affecting a lot of people worldwide. Johns Hopkins University has created a COVID map to show near real-time information of confirmed cases by country, state, or county [11]. In hotspot detection, spatial join and map overlay operation is required between polygonal layer (county boundaries) and Covid-19 disease occurrences (points). Our filters can be used here. In addition, the cellphone can be tracked to know the scope of people's activity [12]. The activity route of a cellphone can be considered as a polyline and the different areas can be considered as polygons.

It is necessary to apply high-performance computing to perform geometric intersection for a huge amount of spatial data. CPU-based geometric intersection

algorithms have been studied in many papers [8, 9, 13]. Since GPU is suited to process compute-intensive work, one objective of this dissertation is to develop and improve GPU-based geometric intersection algorithms. Compiler directives have been used to GPU-based parallelization of computational geometry algorithms earlier [14]. In the OpenACC-based spatial join implementation, it used classical filter and refine technique where filter phase was done on a CPU using R-tree query and refinement phase was done on a GPU. The performance for large data was unsatisfactory when compared to the sequential implementation of spatial join on a CPU. This also motivates the present work.

Chapter 2
Literature Review

2.1 Introduction

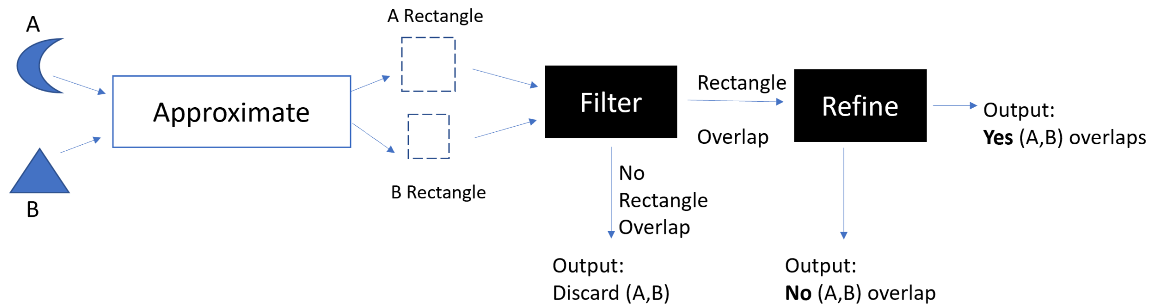


Figure 2.1: Filter and refine technique for one polygon pair

Filter and refine strategy is used in many spatial computing algorithms for spatial query, spatial join, and overlay in geographic information system [15]. It 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 [16]. 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 massively parallel hardware that accelerates the filter and refine computations. Figure 2.1 shows an example of the filter and refine technique. Two polygons are represented by two rectangles. The filter phase will check if these two rectangles overlap or not. If they do, the refine phase will check if these two geometry overlap or not by using their actual line segments. Typically, the filter step is lightweight and the refinement step is compute-intensive because of complex computational geometry algorithms. The

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 [17, 18, 8, 19, 9, 20].

In this dissertation, we mainly study GPU-based implementation of the 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-in-polygon (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 [21, 8, 22, 7, 9]. Depending on the variation in size of line segments and its spatial distribution, planesweep [23] and grid-based algorithms on CPU [24] and GPU [25, 1] have been reported in the literature. In addition, there are data structures like segment tree and R-tree that have been used to speedup LSI problems [26]. Different methods employed in filter and refine based spatial query processing have been discussed in [15]. R-tree is used for building a spatial index and MBR query [27]. Partitioning a polygon by decomposing its area into smaller and simpler geometries has been studied earlier [28]. Existing tools in GIS invoke computational geometry algorithms on shapes made of 2D co-ordinate data. In computer graphics, a complicated shape is often decomposed by triangulation. For efficiency, triangulations are used instead of actual geometry.

Plane-sweep is a fundamental technique in computational geometry and it has been parallelized on multi-core and manycore architectures [23, 29, 14]. Boolean

set operations like union and intersection on polygons require line segment intersections and point-in-polygon test [30]. GPU-based acceleration of segment intersections and point-in-polygon test have been studied in the domain of GIS and spatial databases [21, 31, 32, 4, 6]. GPU-based similarity searches was studied in the paper [33].

Given two input layers of geometries, the filter step uses MBR approximation of a geometry and the refine step uses the actual vertices that represent it. In addition to MBR, other approximations which are used as filters are rotated minimum bounding box, convex hull, minimum bounding circle and ellipse, n-cornered bounding polygon, etc [34, 35, 36, 16]. Convex hull can be used as a replacement for MBR in spatial queries. Douglas-Peucker line simplification algorithm and its variants reduce the number of points to represent a curve [37]. Line simplification is primarily used for visualization on a map. In case of geometries like polygons, using the simplified version may not yield correct results for spatial queries.Recent approaches for implementing spatial computations using GPU, include a variety of filters, for instance, Common MBR Filter (CMF) [38], two-level uniform grid [24], Grid-CMF [39].

2.2 Minimum Bounding Box and Related Algorithms

2.2.1 Clipped Bounding Box

Minimum bounding box (MBB) is the smallest axis-aligned rectangle that encloses the spatial object, such as a polyline or polygon. Generally, it only requires two multi-dimensional points to store MBB and computing the MBB is also simple. It can be applied to the filters of basic spatial operations (such as, the intersection test) because comparing an MBB with other MBBs is cheap. In addition, MBB is the main component of R-tree [40] and its variants [41]. Many filters are also based on MBB [42, 43, 44].

However, a spatial object's MBB may include 'dead space' that does not contain any real objects so it could decrease the filter's efficacy and precision. Therefore, the paper [45] introduces the clipped bounding box (CBB) and proposes to clip away empty corners of MBBs by clip points. The method can prune more area with low overhead and can be plugged into any R-tree variants because the clipped corners are just supplemental to the original MBBs. The method is also general since it can be also applied to more than 2 dimensions and there is no restriction on the data. For example, the data can be lines, planes, and volumetric spatial objects. Since they reduce the coverage by not representing the dead space in the corners, they also reduce the MBBs overlap potentially.

One main component of CBB is the clip point with its bitmask which means the orientation. The bitmask can be 00(left bottom), 01(left top), 10(right top) and 11(right bottom). For example, a point with R^{11} represents an area that is at the top right of the MBB and contains no object. One important property of the clip point is that no object overlaps with the space between the clip point and its relevant corner R^b based on bitmask. In other words, the area should be totally dead space. If we want to test a query region Q with CBB (if it only has one clip point c), we only need to compare Q to the original MBB and then to the clip point c. The calculation is cheap but the improvement should be profound.

Deciding clip points is an important step. There are two approaches: the skyline-only approach and the point-splicing approach. The skyline-only approach is the basic method. The oriented skyline of a set of points with the orientation mask b is the subset of points that are not dominated by other points. (For example, if point p dominates point q and the orientation mask is b, p is closer to R^b on each dimension than q.)

The point-splicing approach can be considered an improvement of the skyline-only approach. To clip away more dead space, they splice the skyline points

and the new points are called stairline points. Splicing points means that it takes the max or min coordinate values of two given points based on the opposite orientation of mask b. For example, a stairline point should clip away more dead space than the skyline points P or Q since the stairline point takes coordinates from P and Q that are farthest from the corner R^b.

Eliminating dead space is only helpful if that is where query rectangle intersect. Therefore, how to apply these ideas to R-tree and its variants is also studied in the paper [45]. The intention of CBB is to make such applications more efficient and precise.

2.2.2 Common Minimum Bounding Rectangle

An algorithm for polygon intersection with N and M line segments requires $O(N \cdot M)$ 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. Computing an MBR and checking an MBR with other MBRs are much cheaper than checking the intersection points between line segments so MBR is widely used in the filters of basic spatial operations, such as the intersection test.

Common Minimum Bounding Rectangle (CMBR) is a method that is based on Minimum Bounding Rectangle of a polygon. For a pair of polygons, the CMBR is an area where the two MBRs overlap. Figure 2.2 shows two examples of CMBR. Black rectangles are two polygons' MBRs and the red rectangle is their CMBR. When two MBRs overlap as shown in Figure 2.2(b), it is possible that these two polygons do not overlap. Such cases can be safely ignored in the refinement phase as described in Chapter 4. This particular case can be detected in the filter phase itself

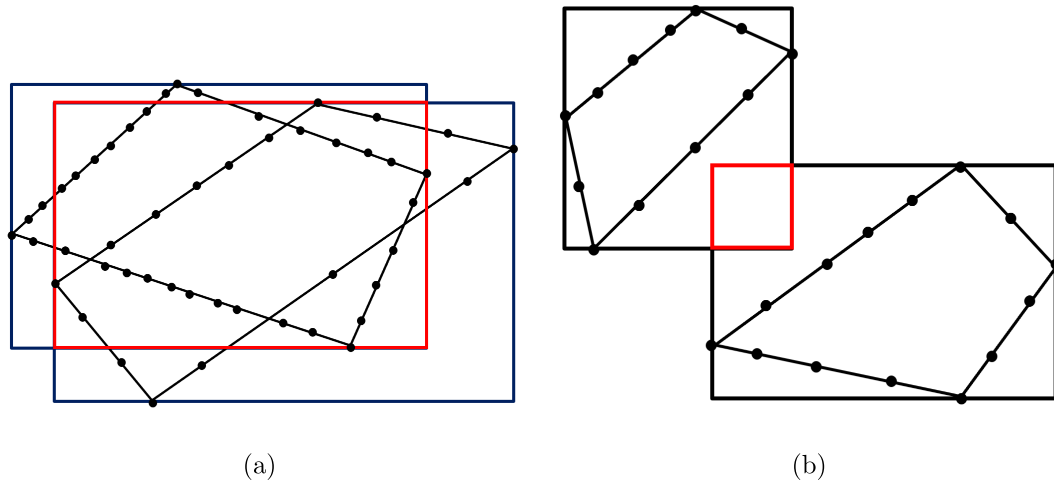


Figure 2.2: Common minimum bounding rectangle examples (Red rectangle is CMBR.)

by checking if the vertices of a polygon lie in the CMBR area or not. Line segments that do not occupy the CMBR area can be safely ignored for LSI function. CMBR of a polygon has been used in earlier work [38] to improve the performance of spatial join on GPU.

In some cases, the common MBR area may contain most of the vertices of the overlapping polygons. This particular case is shown in Figure 2.2(a). This limitation of the CMBR approach makes it less effective [39]. A filter based on $PolySketch$ provides an alternative way of reducing the workload in the LSI function.

2.3 Polygon Intersection Related Algorithms

2.3.1 Traditional Algorithms for Polygon Intersection

There are two categories of existing approaches to calculating the intersection of polygons: numerical algorithms and deterministic algorithms. Generally, numerical algorithms use the Monte Carlo to calculate a definite integral based on the probability theory and create two random numbers for random point's coordinates and check whether the random point not only falls a polygon but also falls into

another polygon. We can get accurate results but the performance is limited by the number of polygons because we need to conduct point inclusion tests for a lot of random points [46]. A deterministic algorithm checks whether each side of a polygon intersects with another polygon. If is true, we check the relationship between the line segments. The relationship can be one endpoint overlap and collinear, parts overlap, completely overlap, intersect with two endpoints, intersection with line, orthogonal or no intersection. Then, we need to check whether the vertex of any polygon is inside another polygon. The paper [46] says it is difficult to apply this method to very large polygons' intersection because of heavy computational load.

Polygon clipping can be a part of polygon intersection. There are many algorithms in computer graphics were designed for 2-dimensional polygon clipping. For example, Greiner-Hormann (GH) algorithm [8] and Vatti [13] algorithm are famous and they can clip arbitrary polygons in a reasonable time. Recently, the paper [9] improves the original GH clipping algorithm by handling the degenerate intersections.

GH polygon clipping algorithm is famous but it can not handle the degenerate intersection cases. Degenerate is that a vertex of a polygon lies on an edge or coincides with a vertex of another polygon or vice versa. This is an important problem so the paper [9] proposes a method to solve it based on the original GH algorithm.

There are three components of the original GH algorithm: Intersection phase, Labelling phase, and Tracing phase. The intersection phase is to get all intersection points firstly and merge them into the data structure. They present all polygons (input as well as output) as doubly linked lists and the intersection point inserted into the subject polygon's data structure is connected to the point inserted into the clip polygon's data structure. For the labeling phase, they trace each polygon once and mark entry and exit points to the other polygon's interior. For

the tracing phase, they start at one of the intersection points, move along based on the rules and get the final output.

The main difference between the new algorithm and the original GH algorithm is that a more refined analysis of the intersection points in the labeling phase. They find the degenerate intersections and handle them correctly by using local orientation tests to mark some intersection vertices as crossing intersections. These points were labeled entry or exit points directly as the GH algorithm.

According to the different types, the algorithm has corresponding methods to insert and link the intersection points. Then, the labeling phase marks the intersection points as crossing or bouncing. In addition, all intersection vertices with adjacent overlapping edges form polygonal intersection chains which will be marked as delayed crossing or delayed bouncing. The tracing phase will generate the output polygons based on the information.

2.3.2 GPU-based Algorithms for Spatial Aggregation

Visual exploration of spatial data is another intersecting topic. It requires spatial aggregation queries which can summarize the data from different areas. Urbane [47] is a visualization framework to explore the real data sets and visualize them according to different resolutions. Urbane can also update the results by changing different parameters dynamically, which becomes challenging when the data set is huge and should be done within a limited time. Such visualizations are helpful for analyzing multivariate data.

An example of spatial data visualization that is the heat map of taxi pick-ups by using neighborhoods and census tracts with Urbane. Spatial aggregation queries are defined as: given two data sets points P and regions R, the result of the query is an aggregation (AGG) over the results of the join between P and R [1]. Sometimes, policy makers are interested in the city to vary with new

zoning or changing the zonal boundaries or construction rules. Such new summarizations should be executed in real-time after changing the configurations.

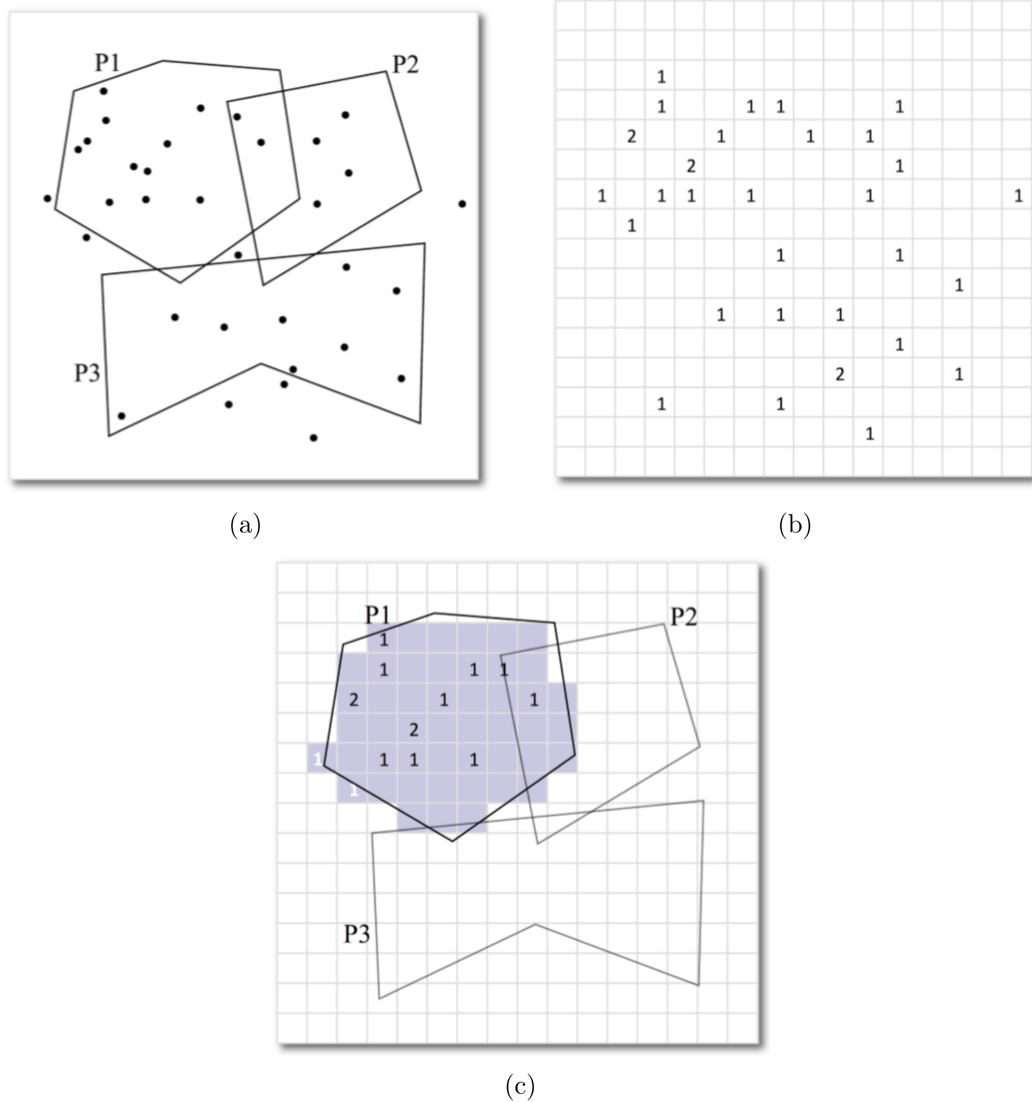


Figure 2.3: An example of raster join approach; (a) input data, (b) all points are rendered onto an FBO, (c) the pixel values are aggregated corresponding to fragments of a polygon [1]

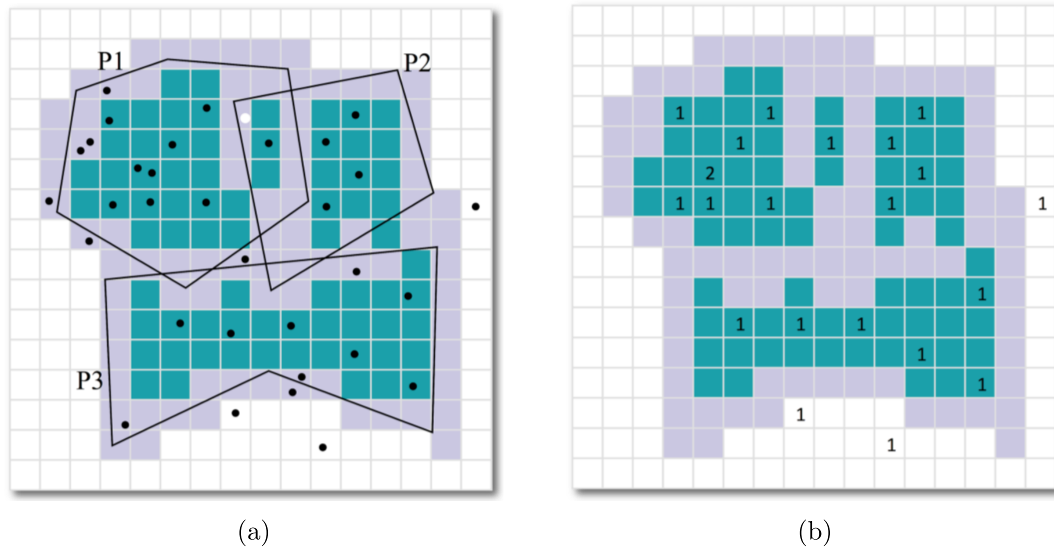
In the paper [1], they use GPU to perform spatial aggregation queries that are the foundation of exploring and visualizing spatio-temporal data. Two new methods were introduced to accelerate the execution of the previous operation. One method is bounded raster join that can get approximate results by drawing

operations. Another one is accurate raster join that can get exact results. There are some challenges of enabling quick response time to the spatial aggregation: (1) Point-in-polygon (PIP) test that requires intensive computation takes linear time according to the sizes of the polygons. (2) GPU memory is limited so the join may be performed in batches, which means that there is a lot of additional data transfer between CPU and GPU. In addition, users may dynamically change filtering conditions, aggregation operations, polygonal regions used in the query, or such operations when they use interactive visual analytics tools.

Since this paper focuses on analytical queries instead of explicit materialization of the join result, they apply the rasterization method to spatial aggregation. Drawing points and polygons are two important steps in their method. They draw the points on a canvas and maintain ‘partial aggregates’ in the cells to keep track of the intersections. Then, they draw the polygons on the same canvas and calculate the final aggregate result with the partial aggregates of the cells that intersect polygon(s).

Figure 2.3(a) is an example of input data that will be used to raster join approach. It contains three polygons and 33 points. Firstly, all points are rendered onto an FBO (Frame buffer objects) that OpenGL uses to output results into a ‘virtual’ screen. It stores the count of points in each pixel as shown in Figure 2.3(b). Then, they render all polygons triangulated [48] and update the query results according to the requirements. The polygons are converted into discrete fragments by using rasterization [49]. Figure 2.3(c) shows that the pixel values are aggregated according to the fragments of one of the polygons. As the author introduced, their bounded raster join is an approximate method that may cause some false results. A pixel will be part of a triangle only when the pixel’s center falls in the triangle. As shown in Figure 2.3(c), the false positive counts are marked in white. One way to solve it is to decrease the pixel size so the pixels can better approximate the object.

According to their result, there is no big difference between the bounded raster join result and the accurate result but bounded raster join took less time.



(a) (b)

Figure 2.4: Accurate raster join: (a) forming boundaries of polygons, and (b) ‘draw’ polygons [1]

To make the method can be applied to any application which may require accurate results, they improve the core raster approach by calculating completely accurate results and call it ‘accurate raster join’. As shown in Figure 2.4(a), the fragments (pixels) with green or white means they are completely inside or outside a polygon. Grid cells with violet mean that they are on the boundary of the polygon(s). Since the wrong result only may occur within ‘violet grid cells’, they propose to perform Point-in-polygon (PIP) test for the points only within these cells to make sure all results correct. Before drawing points and polygons, the accurate raster join draws the outline of all the polygons. It renders the boundaries of the polygons onto an FBO. Then, they draw points and perform a PIP test for every candidate polygon if the fragment corresponding to the point is a boundary pixel. If the fragment is not a boundary pixel, the process is the same as the core approach. After this, they render polygons and the only difference is that they discard the

fragment which is on a boundary pixel during this step. By using OpenGL, their methods are portable and easy to be applied to existing database systems. According to the results, the bounded raster join can get much better performance and accurate raster join is also good. Since the GPU memory is different from CPU memory, their method performs the spatial aggregation in a smart way by drawing operations to decrease data movement. It also reduces the workload in the PIP test. In real applications, it is possible to perform multiple aggregates so their current work should be extended to handle this.

2.4 Spatial Analytics System and Hybrid Platform

2.4.1 CPU-GPU Hybrid Platform

Since the development of data acquisition techniques, the size of spatial data is increasing on a huge scale. Many different real world applications are generating and collecting a lot of spatial data, such as in-vehicle GPS navigation systems or location-based services [50]. It becomes more challenging to query or analyze such a massive amount of spatial data in an efficient way. Spatial proximity queries, window-based queries, spatial cross-matching queries, nearest neighbor queries are some basic geospatial analysis [51]. The combinations of such queries can be used in real world use cases, such as traffic analysis. PostGIS, ArcGIS, and other traditional spatial database management systems support spatial queries but the performance is affected by the lack of an effective spatial partitioning mechanism that can balance the workload and manage compute-intensive operations [52]. MapReduce is able to partition data and distribute them to multiple workers. According to their observations [52], General Purpose Graphical Processing Unit (GPGPU) is suited to handle compute-intensive challenges so they explore the GPU-based spatial crossing-matching operation. Then, a hybrid CPU-GPU approach was developed to speedup spatial cross-matching queries over geospatial datasets. One challenge of

high performance computing is to construct an efficient mechanism to schedule the tasks between GPU and CPU to make the best use of all devices. In addition, there are some challenges of using GPU compared to CPU: slower memory access, data movement cost, lower clock speed, and so on [53].

Generally, there are three phases in a geospatial query: (1) parsing, (2) indexing/filtering, and (3) refining. This is also the filter-refine technique. Parsing is to convert the input data to in-memory spatial data structures that will be used to build a spatial index. Using the indexes can filter the spatial data objects and only keep the operation relevant objects for the later operations. Refining is used to get final results.

In their platform, there are three major components: (1) query tasks, (2) scheduler, and (3) execution engine. The cross-matching workflow was divided into the pipeline of tasks after reading the input data, which makes different query operations can be executed on different processing units. According to their hybrid CPU-GPU framework, GPU and multi-core CPU are the processing units and they assign the tasks to process units according to the tasks' characteristics. Based on their analysis [52], parsing, indexing and filtering are not suitable to be executed on GPU because of their processing dependencies. Therefore, for the previous phases, they execute them on CPU. For indexing/filtering, they use MBBs of polygons to create a Hilbert R-tree index that checks polygons whose MBBs overlap with others. In addition, refining is easier to be parallelized on GPU so the multi-core CPU and GPU can be used to this phase at the same time.

The scheduler that contains task and thread queues is another important component in their system. The function of it is to assign new tasks to available processing workers according to first come first serve (FCFS) or priority queue (PQ). FCFS is simpler because the refiner task will be executed on CPU or GPU according to the availability of the processing worker and the task position. For PQ,

the refiner task will be executed based on their potential speedup gain, which means that it is higher to be executed on GPU when the priority is higher. The execution engine is the last important component of the system. It includes different libraries for actual spatial computation on CPU and GPU.

According to their experiments, for the refinement phase, hybrid (1CPU-GPU) works better than one single thread CPU and the speedup varied between 10x to 11x according to the different number of overlapping polygons. In addition, they also conclude that the data copying overhead is high for GPU computation, which means that users should consider carefully whether to accelerate jobs on GPU according to the potential speedup or when the dataset is large. For the hybrid framework, using more CPUs can lead to better performance, which can reach 17x speedup at most.

2.4.2 Parallel Implementation on CPU-GPU

The paper [52] shows that Graphical Processing Unit (GPU) is suited to handle compute-intensive work so they propose a hybrid CPU-GPU approach to accelerate spatial cross-matching queries. The paper [1] proposes to use GPU to perform spatial aggregation query by the drawing operations. Other papers also implement their algorithms on GPU to handle computing challenges [54, 55, 46]. The scale of polygon data has been increasing tremendously and the rapid growth in the spatial data also increases the requirements of efficiency for spatial analysis. Based on the structure and strong capability of GPU, it is worth performing such operations by using GPU(s). However, there are still some challenges: 1) the cost of copying data between GPU and CPU, 2) reading and writing data in global memory are expensive, 3) race condition, 4) limited GPU memory, and so on. GPU is generally used to provide the images in computer games so it emphasizes high throughput instead of low latency compared to CPU. It has more ALU units and is faster than

CPU. The complex problems should be divided into a lot of separate tasks and GPU gets them done at once, which is similar to process graphics that require to be done at once. Compared to CPU that includes a few cores with a lot of cache memory, GPU contains hundreds or thousands of cores that can deal with a great number of threads simultaneously. Since the structure is different from CPU, programming on GPU is very different. For example, complicated code with lots of if-else branches and irregular program flow (such as, the plane sweep algorithm and R-tree) are harder to be parallelized and get good performance.

Parallelism in multi-core and manycore hardware can be exploited using compiler-based approaches like OpenMP and OpenACC. OpenMP includes a collection of compiler directives (pragma) and library routines used for parallelization of existing C, C++, or Fortran code. OpenACC is a programming standard for parallel computing. It is easier to be learned since the users just need to add some compile directives to the original code (such as, C++). It is also easy to maintain the code. OpenACC is based on CUDA. CUDA is a parallel computing platform and API model for GPU.

2.4.3 Modern Spatial Analytics Systems

With the development of GPS-enabled devices and their applications, spatial data is pervasive. This attracts more research communities to develop related algorithms or systems to process and analyze spatial data efficiently [56, 57, 58, 59, 60]. The paper [2] explores and compares the available modern spatial analytics systems based on their features and queries over real world datasets. It mainly compares five Spark-based systems: SpatialSpark, GeoSpark, Simba, Magellan, and LocationSpark. It also introduces SpatialHadoop and HadoopGIS but does not evaluate them because Hadoop based systems usually perform poorly compared to Spark based systems. In addition, four dada types (points, linestrings, rectangles,

and polygons) and five spatial queries (range query, kNN query, spatial joins, distance join, and kNN join) are used for the evaluation. Range query and kNN query are single relation operations. Spatial join, distance join and kNN join are join operations. There are some other spatial queries: spatial data mining operations, raster operations, and computational geometry operations [61]. However, the evaluated systems do not support them.

A two-level indexing scheme (multiple local indices in the slave nodes and a global index in the master node) is used in distributed systems. Generally, the input data is partitioned and every partition will be indexed based on the spatial index, such as R-tree, Quadtree, and so on. The details of partitioning are introduced in the paper [61].

Hadoop-GIS was the first system to support spatial queries based on Hadoop. The objects are mapped to the tiles after the space is partitioned by 'uniform grid'. The high density tiles would be broken down into smaller tiles. SATO [62] is a partitioning technique used to Hadoop-GIS and it has four steps that are Sample, Analyze, Tear and Optimize. They sample 1-3% of the data and compute the density distribution of the data set. Then, the Analyzer decides the global partitioning scheme for the global partitions and every global partition will be partitioned to create local partitions. SpatialHadoop is also based on Hadoop and native to support spatial data. It partitions datasets based on fitting one HDFS block and the close objects should be assigned to the same partition. It also uses the Sort-Tile-Recursive (SRT) algorithm that computes the number of partitions according to the size of the input data and fills the R-tree.

SpatialSpark, GeoSpark, Magellan, SIMBA, and LocationSpark are based on Apache Spark. SpatialSpark relies on in-memory processing and is simpler than the other 4 systems. GeoSpark is an in-memory cluster computing framework and includes three layers: (1) Apache Spark Layer that supports native functions

supported by Spark (such as, load data to storage); (2) Spatial RDD Layer that efficiently partitions SRDD elements across machines by extending spatial RDDs (SRDDs); and (3) Spatial Query Processing Layer. In addition, GeoSpark provides a geometrical operations library that supports basic geometrical operations, such as find overlapping objects. Magellan is the distributed execution system for spatial data. It also supports geometric predicates, such as within, intersect and contain. Simba that means Spatial In-Memory Big Data Analytics is distributed analytics engine. It also extends Spark SQL to accommodate spatial operations. A spatial RDD layer called Location-RDD that can be cached in memory is introduced by LocationSpark. In addition, there is a query scheduler used in LocationSpark.

To evaluate these Spark-based systems, they deploy variable sized clusters on Amazon AWS and use different types of real data sets to evaluate the systems. According to their analysis, GeoSpark is a more complete spatial analytics system. Compared to other systems, it supports more data types and queries at the same time and has more partitioning techniques. Their results also prove that GeoSpark works better than other systems in most cases. However, it takes more memory to store and partition input datasets and does not support kNN joins. Magellan is another good system that works well for some spatial join compared to other systems but it does not optimize range queries. In addition, it does not support kNN queries, distance joins and kNN joins. LocationSpark includes good query scheduler and optimizer. The spatial bloom filter (sFilter) is used to it and other systems can also incorporate such filters. SpatialSpark takes high execution memory. Furthermore, GeoSpark and Magellan are still actively under development.

Chapter 3
PolySketch: An Efficient Filter for Geometric Intersection Computations using GPU

3.1 Introduction

It has been shown that as geometries are getting larger in size, the refinement phase is taking most of the time [31]. Decreasing the number of candidates produced in the filter phase also reduces the workload in the refinement phase. Therefore, we propose applying a hierarchy of MBR and *PolySketch* filter to improve the filter efficiency. Not all segments of a polygon will intersect with the segments of another polygon. Expensive polygon intersections in the refinement phase can be possibly eliminated by using the sketch of a polygon. Further improvement is possible by the GPU-acceleration of computational geometry algorithms in the refinement phase.

We have extended the classical filter and refine algorithms using PolySketch Filter to improve the performance of geospatial computations. In addition to filtering polygons by their Minimum Bounding Rectangle (MBR), our hierarchical approach explores further filtering using tiles (smaller MBRs) to increase the effectiveness of filtering and decrease the computational workload in the refinement phase. The basic idea is to represent a large geometry using its sketch that is made up of a collection of tiles. Each tile is a subset of contiguous vertices with the corresponding MBR induced by the subset. Our *PolySketch* filter not only reduces the candidate pairs but also reduces the workload in the refinement phase. PolySketch filter reduces on average 99.77% of the workload of finding line segment intersections.

3.2 Related Work

Theoretical PRAM algorithms and multi-threaded implementations for polygon clipping have been designed [22, 7]. The intersection of two cross-layer polygonal MBRs (CMBR) was used earlier in the GPU-based spatial join system called GCMF [38] to filter out candidate pairs that do not need further refinement. CMBR is effective in cases where it can filter out the majority of the line segments. This leads to a reduction in workload. It was observed that in some cases CMBR was not effective in workload reduction. So, the CMBR technique was further improved by creating grid inside the area of CMBR for further filtering [39]. As opposed to CMBR, *PolySketch* is a hierarchical technique. Other approaches used in literature include *PixelBox* where geometries represented as co-ordinates are converted to a raster format (pixels) to leverage image processing using a GPU [52]. CGAI package [63, 64] provides an algorithm called 'Intersecting Sequences of dD Iso-oriented Boxes', which can detect all intersections for polyhedral surfaces by using iso-oriented boxes. Our *PolySketch* system can employ the CMBR technique in a hierarchical manner to weed out the pairs of cross-layer tiles that do not need further refinement as described in Chapter 4.

Spatial partitioning of geometries using techniques like uniform grid [6, 65], quadtree, and binary space partitioning has been studied in literature. In our system, we do not explicitly do spatial partitioning. Instead, we use data partitioning by tiling. Tiling induces spatial partitioning. We do not use uniform or adaptive grid partitioning of input layers of spatial data.

3.3 Geometric Intersection Computations

3.3.1 Problem Description

With two layers R and S as inputs, the output of spatial join is a collection of pairs (r, s) such that $r \in R$, $s \in S$, and r and s overlap spatially. Overlap is one of the spatial relations between r and s. Other spatial relations are *Intersects*, *Touches*, *Contains*, etc. Spatial databases and GIS support such topological relations on two layers of geometries.

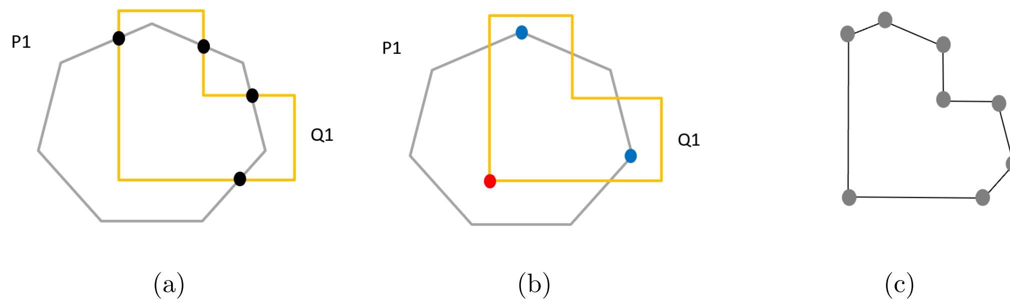


Figure 3.1: (a) Line segment intersection vertices, (b) Vertices inside another polygon, and (c) Output polygon (Best Viewed in Color)

The refinement phase involves computational geometry algorithms on a variety of shapes. Computing the topological relations and geographic map overlay requires two kernels namely, line segment intersection (LSI) and Point-in-Polygon (PNP). An example of polygon intersection is shown in Figure 3.1. There are two overlapping polygons P1 and Q1. The first step is to find line segment intersection vertices (black) among line segments from the two polygons as shown in Figure 3.1(a). The second step requires the PNP function to find polygon vertices that are inside another polygon e.g., one vertex (red) of Q1 is inside P1 and two vertices (blue) of P1 are inside Q1 as shown in Figure 3.1(b). Finally, output polygon(s) are produced by combining the output of LSI and PNP functions. An

output polygon is shown in Figure 3.1(c).

In spatial join, the output polygon produced by polygon intersection is not required. Spatial join is based on a boolean predicate, e.g., *Intersects*. *Intersects* predicate returns true if two polygons have line segment intersection or a polygon is inside another polygon. In polygon overlay, output polygon needs to be computed as well. This requires finding all line segment intersections as well as vertices of the polygon that are inside another polygon. Because of this difference, polygon overlay is computationally more expensive than spatial join. In our work, we compute all the segment intersections and the vertices of a polygon that are inside another one. The number of segment intersections (can be quadratic in the worst case) is variable for each candidate pair, so handling it on GPU either requires redundant computation because of counting the number of intersections a priori or using atomic locks while storing them.

3.3.2 PolySketch Algorithm

Listing 3.1: Tile Data Structure

```
struct Tile {
  int start, end;
  vertex *v; // vertex array
  MBR mbr; //MBR of v[start] to v[end]
};
```

Sketch of a polygon/polyline is made by tiling its boundary in such a way that one tile represents an MBR of the vertices in that tile. These tiles are contiguous and two adjacent tiles share a vertex. A sketch is designed as a lightweight representation to be used in the filter phase of a filter-and-refine algorithm in a spatial computation, e.g., join, overlays. Figure 3.2 shows the

hierarchical tiling approach. A tile is defined as a C structure. When compared to MBR of a geometry, a sketch of a geometry has less dead-space. As such, better filter efficiency is possible at the cost of additional space requirements.

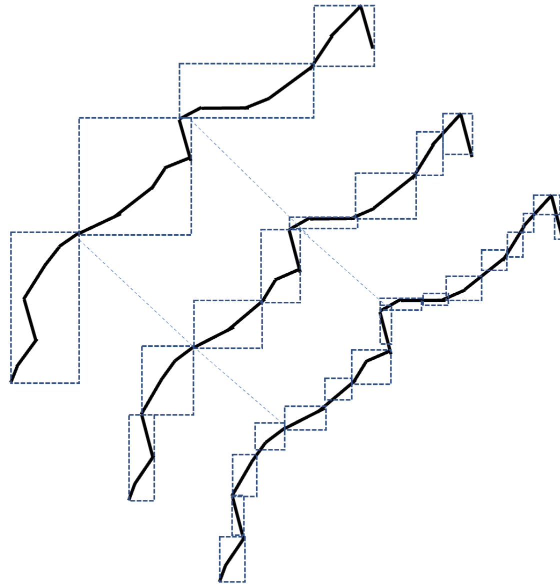


Figure 3.2: PolySketch of polyline data composed of tiling at three different levels.

A PolySketch for a geometry with n vertices and tile length set as b consists of $\lceil \frac{n}{b} \rceil$ tiles. Since, in each tile, an MBR for the vertices in that tile needs to be computed, building a sketch of a polygon is $O(n)$ operation. An MBR of an entire geometry is also $O(n)$ operation. As will see later in the experimental results, sketching provides a space-time tradeoff because of its hierarchical nature.

A tree can be constructed to represent a hierarchy of PolySketches at different levels. Using the leaf-level tiles, internal nodes of the tree can be constructed using union of two successive tiles. An MBR of a polygon can be thought of as its level 0 sketch with its start index as 0 and end index as the number of vertices in the polygon. For a polygon with N vertices, there are O(log N) sketches possible in a tree-based representation. However, it suffices to use a few levels only as shown in Figure 3.2 for space-efficiency.

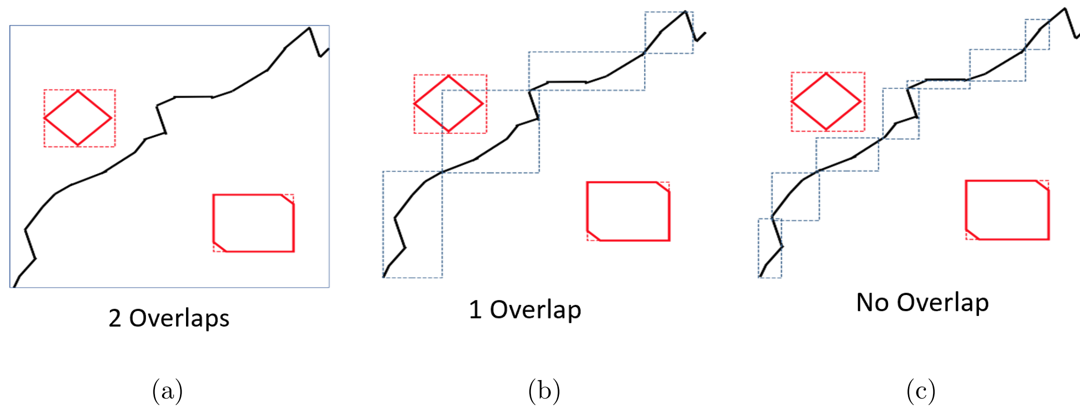


Figure 3.3: Improve approximation using PolySketch

As shown in Figure 3.3(a), the black line is from one layer and two red polygons are from another layer. We will get two candidate tasks by using their MBRs. Using PolySketch and a small tile size as shown in Figure 3.3(c), there is no overlap and we do not need to perform the refine step for them. Therefore, PolySketch can minimize dead space and discard more candidate tasks compared with using the standard R-tree.

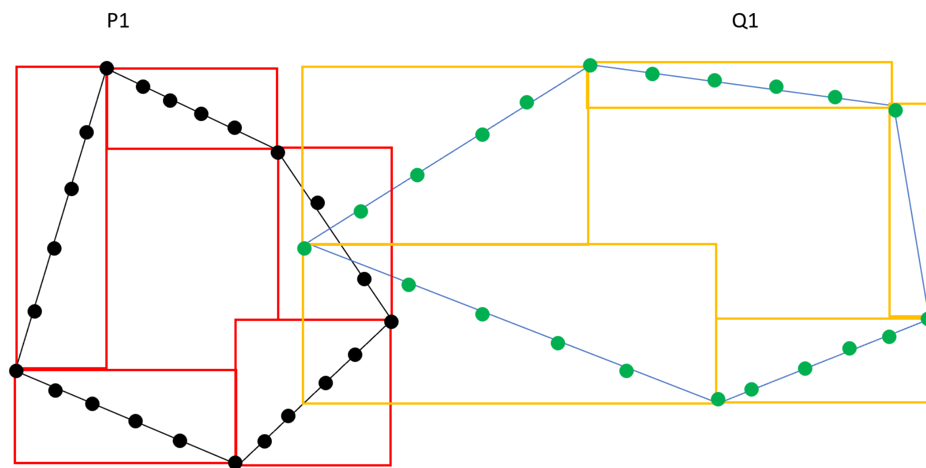


Figure 3.4: Polygon intersection using PolySketch. Tiles for each polygon is shown in different colors.

As shown in Figure 3.4, there are two polygons P1 from layer1 and Q1 from

layer 2. The tile-size is set as five line segments. Q1 consists of twenty-one line segments, so it is divided into five tiles. For polygon intersection between P1 and Q1, we first check if their corresponding tile-MBRs overlap or not. If some tile-MBRs from P1 and Q1 overlap, we record those tile pairs and use the LSI function for those pairs. If there is no tile-MBR overlap, we discard this task for the LSI function. In Figure 3.4, we can see that there are three pairs of tiles that have overlap. A tile located in the lower left corner of Q1 overlaps with two tiles located in the right-side corner of P1. Similarly, another tile located in the upper left corner of Q1 overlaps with one tile located in the upper right corner of P1. Other tiles and their corresponding vertices can be safely ignored in the LSI function.

Checking if two tile-MBRs overlap is computationally cheaper than finding the segment intersection between two line segments. In one of the datasets that we have used, about 13% of polygons have more than 500 vertices. Since, a tile's MBR contains a fraction of the vertices of a polygon, using it in place of actual vertices in the filter phase is a cost-effective strategy. Algorithm 1 shows how to apply *PolySketch* Filter and Refine for polygon intersection tasks using compiler directives supported by OpenACC.

3.4 Experimental Setup and Results

3.4.1 Datasets

We have used three datasets to evaluate our system: (1) Urban, (2) Water, and (3) Lakes. The details are shown in Table 3.1. 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/datas ets.html.

Algorithm 1 Segment Intersections using PolySketch Filter

```
1: #pragma acc data copyin(layer1Polygons, layer2Polygons) copyout(line segment
   intersections)
2: #pragma acc parallel
3: #pragma acc loop
4: for each taskID ∈ taskArray do
5:     get polygon pair (p,q) using taskID
6:     #pragma acc loop
7:     for each tile t_p ∈ p.tiles do
8:         Calculate t_p.MBR
9:     end for
10:    #pragma acc loop
11:    for each tile t_q∈ q.tiles do
12:        Calculate t_q.MBR
13:    end for
14:    #pragma acc loop reduction (numSegIntersections)
15:    for each tile t_p ∈ p.tiles do
16:        #pragma acc loop
17:        for each tile t_q ∈ q.tiles do
18:            if (t_p.MBR overlaps t_q.MBR) then
19:                Call LSI(t_p.segments, t_q.segments)
20:                #pragma acc atomic
21:                store segment intersections
22:            end if
23:        end for
24:    end for
25: end for
```

Table 3.1: Three real datasets used in our experiments

Label	Dataset	Polygons	Segments	Size
Urban	*ne_10m_urban_areas*	11,878	1.1M	20MB
	ne_10m_states_provinces	4,647	1.3M	50MB
Water	*USA_Water_Bodies*	463,591	24M	520MB
	USA_Block_Boundaries	219,831	60M	1300MB
Lakes	*Lakes*	7.5M	277M	9GB
	Sports	1.8M	20M	590MB

3.4.2 Hardware Description

We have used Intel Xeon E5-2695 multi-core CPU with 45MB cache and base frequency of 2.10GHz. We have used two different kinds of GPU to run the experiments, namely, Titan V and Titan Xp. Titan V is a more powerful GPU and its architecture is NVidia Volta. It has 640 Tensor Cores, 12 GB HBM2 memory, 5120 CUDA Cores, and its memory bandwidth is 652.8GB/s. The architecture of Titan Xp is Pascal. It has 12 GB GDDR5X memory, 3840 CUDA Cores, and its memory bandwidth is 547.7GB/s. For experiments on a single GPU, we have used Titan V. When using multi-GPUs, we used one Titan V and three Titan Xp. The PGI compiler version is 18.10.

3.4.3 Performance of PolySketch Compared to CMBR

For analysis, let us consider that each task has two polygons; P from layer 1 and Q from layer 2. For one task, P has m line segments and Q has n line segments. For the LSI function, every line segment from P should be compared with all line segments from Q. The workload is $m * n$ for every task. Therefore, the total workload for the LSI function is the summation of the workload of individual tasks. The Application of CMBR or *PolySketch* filter decreases the line segments in each task. This leads to workload reduction.

Tables 3.2, 3.3, and 3.4 show *PolySketch* and CMBR's effect on reducing the workload and the number of line segments for the LSI function for different datasets. (In the tables, Sketch means *PolySketch*) We can see that both CMBR and *PolySketch* can reduce a lot of line segments and the workload. *PolySketch* works better than CMBR overall to reduce the total workload which is directly related to the run-time. Therefore, we can get better execution time results by using *PolySketch*.

Another advantage of *PolySketch* is that it is easier to implement using compiler directives. We only need to record overlapping tiles. Therefore, we implemented both *PolySketch* and LSI function together using OpenACC. For implementing CMBR, we need to calculate their CMBRs, test and store the line segments that overlap with the CMBRs. We implemented CMBR to preprocess data on CPU and run the LSI function on GPU.

To be fair, we show CMBR time and its LSI function time separately for CMBR Filter so that we can see how much time the LSI function took. We show execution time for *PolySketch* construction and LSI function together for *PolySketch* Filter. For the bigger data, *PolySketch* method's time which includes *PolySketch* time and its LSI function time is better than the CMBR method's LSI function time which does not include CMBR time. For other data sets, *PolySketch* method's times are similar to the CMBR method's LSI function times which do not include CMBR time.

Table 3.2: Sketch and CMBR effect on the LSI function for Water dataset

Water	No filters	With CMBR	With Sketch
Time(s)	10.47	10.36 + 4.53	1.39
workload	411,876,982,358	16,327,012,938	1,789,226,826
# of segments (L1)	1,036,879,194	26,844,066	242,685,263
# of segments (L2)	1,996,217,931	30,765,554	145,134,707
# of tasks	1,020,458	274,283	321,658

Table 3.2 shows the result for the Water dataset. We can see that *PolySketch* is more effective in reducing the workload compared to CMBR. As we mentioned before, for some polygon pairs, their CMBRs can be very large. This leads to less effective filtering of line segments in those CMBRs, which in turn increases the workload in the refinement phase.

If we only consider the number of line segments present in each individual layer after the application of the CMBR filter, we can see that CMBR is quite

effective in this scenario. This is due to the fact that when the overlap area between two polygons is small, their CMBR will have fewer line segments.

When we consider line segment reduction in a single layer case, *PolySketch* is less effective, even though it is quite effective in workload reduction compared to CMBR. This discrepancy can be explained by the way we count the number of line segments in a tile after the filter phase. For *PolySketch*, since one tile of a polygon may overlap with more than one tile of another polygon. Therefore, when counting the number of line segments in a tile after using *PolySketch* filter, we count those line segments more than once. However, to calculate the workload, we need to consider the line segments in all the candidate pairs from both layers. The workload in the LSI function directly affects the execution time. Table 3.2 also shows that the execution time of using *PolySketch* + LSI function is even shorter than the execution time of the CMBR + LSI function.

Table 3.3: Sketch and CMBR effect on the LSI function for Urban dataset

Urban	No filters	With CMBR	With Sketch
Time(s)	0.4	0.23 + 0.03	0.06
workload	6,453,160,088	25,737,640	7,489,801
# of segments (L1)	3,497,270	914,074	834,146
# of segments (L2)	65,476,891	78,492	847,581
# of tasks	28,687	8,166	9,729

Table 3.4: Sketch and CMBR effect on the LSI function for Lakes dataset

Lakes	No filters	With CMBR	With Sketch
Time(s)	2.20	9.4 + 0.51	1.17
workload	29,289,344,523	260,210,378	37,464,000
# of segments (L1)	1,932,905,302	4,061,067	7,716,460
# of segments (L2)	76,801,765	1,763,838	6,143,938
# of tasks	692,435	132,888	201,107

Table 3.3 shows the results for the Urban dataset. We can see that

PolySketch works better than CMBR in reducing the total workload. For *PolySketch* method, the run-time which includes *PolySketch* time and LSI time is similar to the time of the LSI function after using CMBR Filter. Table 3.4 shows the results for the Lakes dataset. For Lakes dataset as well, *PolySketch* reduces a considerable amount of workload compared to CMBR.

One of the intentions of these two filters is to discard the invalid tasks for the LSI function so we can use GPU efficiently only for the valid tasks where two polygons may have line segment intersection(s). To see our filter's efficiency in discarding invalid tasks for the LSI function, we define its efficiency as

$$D_{\text{Task}} = \frac{\text{The number of tasks discarded}}{\text{The original number of tasks}} \tag{3.1}$$

Table 3.5: Sketch and CMBR effect on reducing tasks of the LSI function for different datasets

	Water	Urban	Lakes
CMBR	73.13%	71.53%	80.81%
Sketch	68.48%	66.09%	70.96%

Table 3.5 shows CMBR and *PolySketch* efficiency percentage for discarding invalid tasks. We can see that both CMBR and *PolySketch* can discard most of the tasks for the LSI function and CMBR is more effective in comparison. However, we use *PolySketch* to reduce tasks as well as workload. This is due to the fact that our compiler directive based CMBR filter implementation is slower compared to *PolySketch* filter implementation.

For quantitative evaluation, here we describe the equations for workload and line segment reduction. In the equations below, C is the candidate set (task), for a candidate pair (i,j), E_i and E_j are the number of the line segments in ith and jth polygons. The symbols with hat notation show the reduced number of line segments

due to hierarchical filtering. Using these symbols, we define the workload reduction percentage and line segment reduction percentage as

$$RP_{\text{Workload}} = \left(1 - \frac{\sum_{(i,j)\in C} \left|\widehat{E}_i\right| * \left|\widehat{E}_j\right|}{\sum_{(i,j)\in C} |E_i| * |E_j|}\right) * 100\% \tag{3.2}$$

and

$$RP_{\text{Line-Segment}} = \left(1 - \frac{\sum_{i\in C} \left|\widehat{E}_i\right|}{\sum_{i\in C} |E_i|}\right) * 100\% \tag{3.3}$$

Table 3.6: Sketch effect on reducing workload and line segments by the LSI function for three datasets using percentage.

Sketch for LSI Function	The workload reduction percentage	The segments reduction percentage for L1	The segments reduction percentage for L2
Water	99.57%	76.59%	92.73%
Urban	99.88%	76.15%	98.7%
Lakes	99.87%	99.6%	92%

Table 3.7: CMBR effect on reducing workload and line segments by the LSI function for three datasets using percentage.

CMBR for LSI Function	The workload reduction percentage	The segments reduction percentage for L1	The segments reduction percentage for L2
Water	96.04%	97.41%	98.46%
Urban	99.6%	73.86%	99.88%
Lakes	99.11%	99.79%	97.7%

Tables 3.6 and 3.7 show the effect of *PolySketch* and CMBR Filter in reducing workload and line segments of each layer for the LSI function for three datasets. *PolySketch* Filter also reduces the number of line segments from both layers. In some cases, it can discard more line segments from a layer where CMBR

Filter is not so effective. In addition, *PolySketch* Filter can also reduce more workload compared to CMBR which is more related to the execution time.

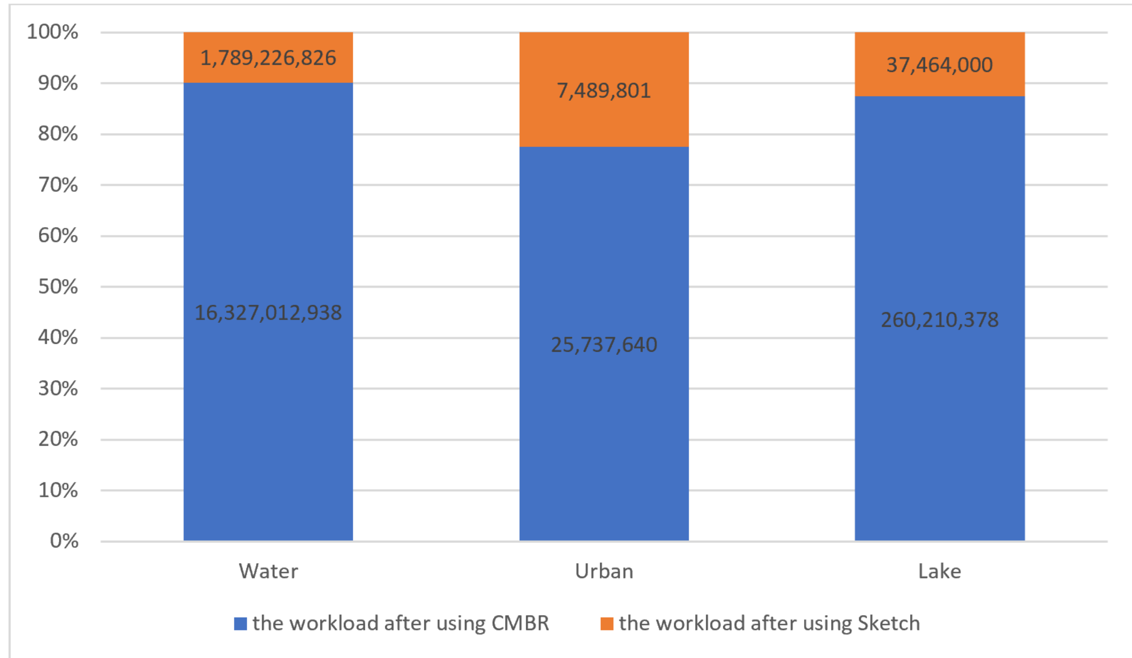


Figure 3.5: The workload in LSI function after using CMBR or Sketch.

Figure 3.5 shows the workload for the LSI function after using CMBR or *PolySketch* Filter. As we can see that *PolySketch* Filter works well in reducing more workload compared with CMBR Filter. For the Water dataset, we can see that the workload after using *PolySketch* is 11% of the workload after using CMBR. For the Urban dataset, the workload after using *PolySketch* is 29% of the workload after using CMBR. For the Lake dataset, the workload after using *PolySketch* is 14% of the workload after using CMBR.

For the Water dataset, the number of thread blocks chosen by PGI compiler was 65535 and the number of threads in a block was 128 for *PolySketch* with the LSI function. Even when the number of tasks was greater than 65535, PGI compiler generated the grid with 65535 thread blocks.

3.4.4 Performance of Using Different PolySketch Sizes

The real-world datasets include small as well as large polygons. Finding an ideal tile-size for a *PolySketch* Filter is difficult. Tile-size of a sketch means the number of line segments in a tile for *PolySketch* Filter algorithm.

Table 3.8: The performance of using different tile-size for Water dataset

Water	Current workload	Current tasks	Time(s)
15 (5)	1,789,226,826	321,658	1.39
15 (10)	1,875,845,026	340,303	1.49
15	1,970,120,151	355,172	1.55
20 (10)	2,554,936,706	356,818	1.62
20	2,772,593,129	385,327	1.82
30	4,460,548,392	442,279	2.29

Table 3.8 shows the performance of using different tile-size for the Water dataset. We can either use the same tile-size for both polygons or use different tile-size. Based on our experience, we recommend different tile-size for small and large polygons in a task. For example, in our experiments, we set tile-size as 15 for large polygons and 5 for small polygons. In our experiment, we found that if the number of line segments of a polygon is smaller than 400, the tile-size of 5 worked well. For larger polygons, the tile-size of 15 worked well in reducing the workload and execution time. As shown in Table 3.8, in the first column, the first number is a tile-size. If there is a bracket, it means we used two tile-sizes and the number in the bracket is the tile-size used for the small polygon. The second column shows the current workload for the LSI function after using *PolySketch* Filter. The third column shows the number of valid tasks for the LSI function after using *PolySketch*. The fourth column is the execution time of running *PolySketch* and LSI function together. We can see that a smaller tile-size works better. It can reduce more workload and discard more invalid tasks. The execution time is also

less. In general, if the tile-size is small enough, we can discard more tile overlap pairs and only use the LSI function for the overlapping tile pairs. Finally, for Water and Lakes, we set tile-size as 15 for large polygons and 5 for small polygons. For the Urban dataset, we set the tile-size as 10.

3.5 Conclusion

We have introduced a new filtering technique based on PolySketch concept which can speedup the LSI function used by spatial join and map overlay computations involving large polygons and polylines. *PolySketch* is a recursive tiling of polygon boundary only; the interior of a polygon is disregarded altogether. As such, the sketch of a polygon does not work in the same way as a polygonal MBR or classical polygon partitioning because *PolySketch* approximates the boundary only. In short, *PolySketch* is not a replacement for MBR in classical filter and refine scenario. It must be used in conjunction with the point-in-polygon test (PNP) to implement standard *intersects* predicate in spatial join.

Creating a PolySketch is a linear time operation, on the other hand, convex hull algorithms require $O(n\log n)$ time. In addition to representing geometries using MBRs, convex and concave hull can also be used. *PolySketch* is similar to the hierarchical bounding technique used for line-curve intersection and curve-curve intersection [66]. We have implemented this filter and refine technique by using OpenACC. *PolySketch* can be constructed in a hierarchical manner efficiently on GPU.

Chapter 4
Hierarchical Filter and Refinement System over Large Polygonal Datasets on CPU-GPU

4.1 Introduction

In this chapter, we introduce our hierarchical filter and refinement technique that we have developed for parallel geometric intersection operations involving large polygons and polylines. The inputs are two layers of large polygonal datasets and the computations are spatial intersection on a pair of cross-layer polygons. These intersections are the compute-intensive spatial data analytic kernels in spatial join and map overlay computations. Spatial join processing was also studied in the paper [67].

When two layers of geometries are overlaid or superimposed in a geographic map, there can be millions of candidate pairs whose MBRs have overlap and need further refinement using computational geometry algorithms. Even though there are PRAM based parallel algorithms available in literature [22, 7], optimal $O(n\log n)$ algorithms for geometric intersection are not available on GPUs. On CPUs, sequential plane-sweep based algorithms are used. For practical parallel implementations, naive $O(n^2)$ algorithms or grid partitioning are used [6, 52]. Even on massively parallel hardware, the quadratic runtime of the naive algorithms results in unacceptable high latency [14]. Grid partitioning may not handle skewed data efficiently. Moreover, partitioning polygons in a uniform or adaptive grid has the disadvantage of a polygon spanning multiple grid cells, thereby increasing redundancy due to duplication of geometries across grid lines.

For the original data, we do not use spatial partitioning using grids. Our approach is to use a combination of filter techniques to reduce the workload in the refinement phase. The input to CMBR and *PolySketch* Filters is the list of

candidate pairs produced by querying R-tree data structure built using the MBRs of the input geometries. We refer to these candidate pairs as tasks because we process them concurrently on a GPU.

A collection of filters applied in a hierarchical manner for speeding up geometric intersection on GPUs was called as Hierarchical Filter and Refine (HiFiRe) technique. 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.

We have implemented this filter and refine technique by using OpenMP and OpenACC. After using R-tree, on average, our filter technique can still discard 69% of polygon pairs that do not have segment intersection points. PolySketch filter reduces on average 99.77% of the workload of finding line segment intersections. PNP based task reduction and Striping algorithms filter out on average 95.84% of the workload of Point-in-Polygon tests. Our CPU-GPU system performs spatial join on two shapefiles, namely USA Water Bodies and USA Block Group Boundaries with 683K polygons in about 10 seconds using NVidia Titan V and Titan Xp GPU.

Table 4.1: Run-times by using different GPUs without filters

		Water	Urban
Titan V	LSI function (s)	10.47	0.4
	PNP function (s)	33.44	0.99
	Total time (s)	43.91	1.39
Titan XP	LSI function (s)	22.16	0.82
	PNP function (s)	92.99	2.51
	Total time (s)	115.15	3.33

4.2 Motivation

First, we used classical filter and refine technique using a CPU-GPU system for LSI and PNP functions accelerated by OpenACC pragmas. R-tree is used for filtering

on a CPU. Table 4.1 shows the results. Even with a powerful GPU, it takes about 44 seconds in total for the larger dataset. This is the motivation behind developing a hierarchical filter and refinement system.

4.3 Hierarchical Filter and Refinement System

4.3.1 System Design Overview

Algorithm 2 Hierarchical Filter-based Segment Intersection

1: Input: Two polygon layers L1 and L2
2: Build R-tree using MBRs of polygons from L1
3: tasks ← Rtree Query using MBRs of polygons from L2
4: newTasks ← Apply CMBR Filter on tasks
5: Apply PolySketch Filter for each newTask
6: **Refine**: Line Segment Intersection Function

In this section, we present how our hierarchical filter and refinement system works. Given two layers of polygons, Algorithm 2 shows the order of application of different filters to find cross-layer line segment intersections in the refine phase. At first, we check which polygon's MBR overlaps with others by using R-tree. If some MBRs overlap, we store these polygon pairs as tasks (T) and every task has two polygons. Then we use CMBR Based Task Reduction Algorithm to check which tasks are valid tasks (T1) or invalid tasks (T2) for the LSI function. After this, we fix the tile-size for creating tiles for the valid tasks (T1) and use $PolySketch$ Filter to reduce the number of line segments and workload. In the last step, we use the LSI function to detect the tasks where two polygons have line segment intersection(s).

We illustrate the benefit of our new approach using an example that shows how hierarchical filtering reduces the overall workload. Let us consider we have two layers of polygons L1 and L2. L1 and L2 consist of 310 and 500 polygons respectively. Table 4.2 shows nine candidate pairs and each pair has two polygons

Table 4.2: Candidate pairs before hierarchical filtering

	1	2	3	4	5	6	7	8	9
L1	P3	P3	P21	P24	P88	P88	P99	P236	P300
	35	35	199	652	998	998	152	4652	52
L2	Q5	Q7	Q56	Q3	Q5	Q12	Q5	Q5	Q457
	65	22	659	832	65	529	65	65	1526

Table 4.3: Eliminating line segments by hierarchical filtering

	1	2	3	4	5	6	7	8	9
L1	P3	P3	P21	P24	P88	P88	P93	P236	P427
	35	0	156	0	0	451	112	324	52
L2	Q5	Q7	Q56	Q3	Q5	Q12	Q5	Q5	Q637
	0	0	34	0	65	256	32	30	0

(P and Q) from L1 and L2 whose MBRs overlap with each other. The number of vertices in a polygon is also shown below the polygon ID. As shown in Table 4.3, the application of *PolySketch* Filter eliminates some line segments that can be safely ignored from further refinement. Moreover, in case of polygons from a few of the candidates, MBR-based Filter eliminates all the line segments. As a result, we can discard the pairs that have zero line segments after applying hierarchical filtering. Since, polygon intersection with n and m line segments is an $O(n * m)$ time algorithm, reducing the line segments by filtering is beneficial.

PNP operation for Polygon: There are some cases when a polygon is contained inside another polygon completely. In spatial database, these cases belong to the *within* or *contains* spatial relations. Finding if a point is inside another polygon is $O(n)$ operation because all the n segments of the polygon need to be examined. As such, a brute-force check for an entire polygon is quadratic in the number of vertices of a polygon. When a polygon A is contained inside another polygon B, then MBR of A is also contained inside MBR of B. However, the reverse is not always true. Our algorithm for PNP can detect these *contains* relation without resorting to a quadratic algorithm. Our algorithm can also detect those

cases where MBR of A is contained inside MBR of B, but A and B are disjoint polygons. As a result, our system can safely ignore these tasks from the expensive refinement operation later. Identifying these cases correctly is possible because we take advantage of the CMBR filter for optimizing PNP operations as well. More details for the PNP-based Task Reduction Algorithm are in Subsection 4.3.4.

Now we will discuss the hierarchy of filters that our system employs to reduce the number of tasks and overall workload.

4.3.2 CMBR Filter Based Task Reduction

The input to this filter is the list of candidate pairs generated by R-tree queries. Each candidate consists of a pair of MBR-overlapping geometries. However, by virtue of CMBR Filter [38], those cross-layer pairs of polygons whose Minimum Bounding Rectangles (MBRs) intersect but their rectangular intersection does not contain line segments from both layers can be safely ignored because those pairs will not have segment intersections. We call such tasks invalid tasks and save computation time by discarding them from further processing in the LSI function. An example of an invalid task is shown in Figure 2.2(b).

Our system classifies the pairs that have line segments inside or across CMBR from two cross-layer polygons as valid tasks; both polygons have segments that are contained inside or across the CMBR boundary. We store this task because the polygons can potentially intersect and need further refinement. An example of a valid task is shown in Figure 2.2(a).

CMBR Filter works well in eliminating some tasks from further refinement. In Subsection 3.4.3, we compare the execution time performance and filter efficiency of using *PolySketch* vs CMBR w.r.t. LSI function.

4.3.3 Workload Reduction by PolySketch Filter

As shown in Figure 3.4, a polygon can be represented as a collection of tiles. The tile-size is user-defined. Intersection of two polygons can be expressed as intersection of their PolySketches. Since, tile-MBR in a *PolySketch* captures the actual area covered by line segments in that tile, line segment intersection can be carried out in two phases: 1) filter phase where tile-MBRs are used for intersection test and 2) refine phase where we only consider the line segments from those tiles that have overlap in LSI function. This is the essence of *PolySketch* Filter.

In CMBR Filter, we need to compare all segments inside the CMBR of one polygon with all segments inside the CMBR of another polygon. If CMBR is large as shown in Figure 2.2(a), we cannot decrease a lot of segments from both polygons, which affects workload in LSI function. However, by using *PolySketch*, a line segment in tile A needs to be compared against the segments of only those tiles which overlap with tile A.

There are certain scenarios where two polygons overlap but it is not detected by the LSI function. Therefore, we use the PNP function for further filtering of tasks.

4.3.4 PNP Based Task Reduction Algorithm

In this algorithm, we find out the vertices of a polygon that are contained inside another polygon. This is required to construct the output polygon for each task. We also find out those tasks where one polygon is entirely inside another polygon in an optimized way. These tasks result in valid output pairs for spatial join or polygon overlay operation. We also want to discard those tasks where polygons are disjoint so that we do not have to invoke quadratic PNP tests for an entire polygon. Figure 4.1 shows the flow chart of this algorithm.

PNP based algorithm is used after the LSI Function. The intention of this algorithm is to discard invalid tasks for the PNP function, divide valid tasks into two different types, do some pre-processing steps and use appropriate PNP functions for them. Before we use LSI Function, we use CMBR Filter to preprocess data to discard some invalid tasks for the LSI function. However, we cannot discard these tasks for the PNP function. We need to check all tasks (T) for the PNP function.

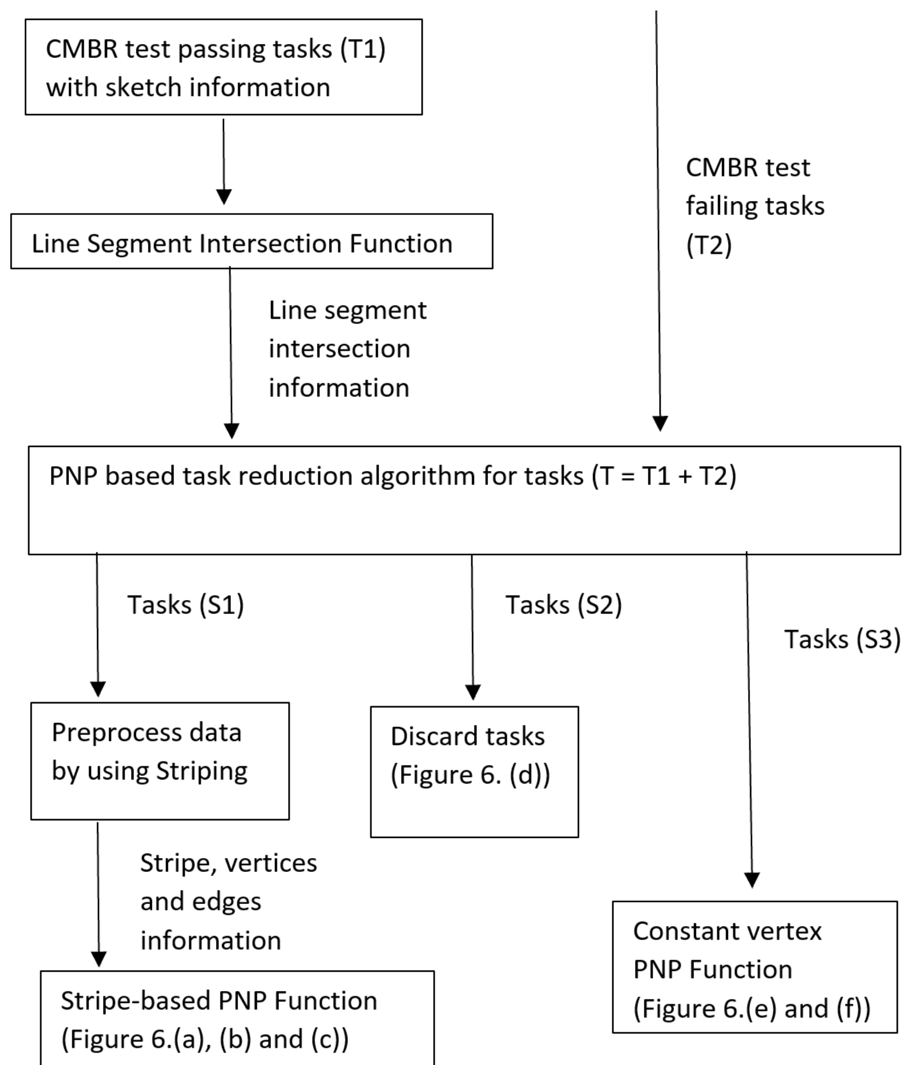


Figure 4.1: Classifying PNP tasks after CMBR Filter

Based on the output of CMBR Filter results, we classify the overall tasks so that we can treat them differently in order to reduce the PNP computation time.

We divide all tasks (T) into three different types of tasks (S1, S2, and S3). S1 includes the tasks where two polygons have line segment intersections. S2 includes the tasks where two polygons do not have line segment intersection and guaranteed not to intersect.

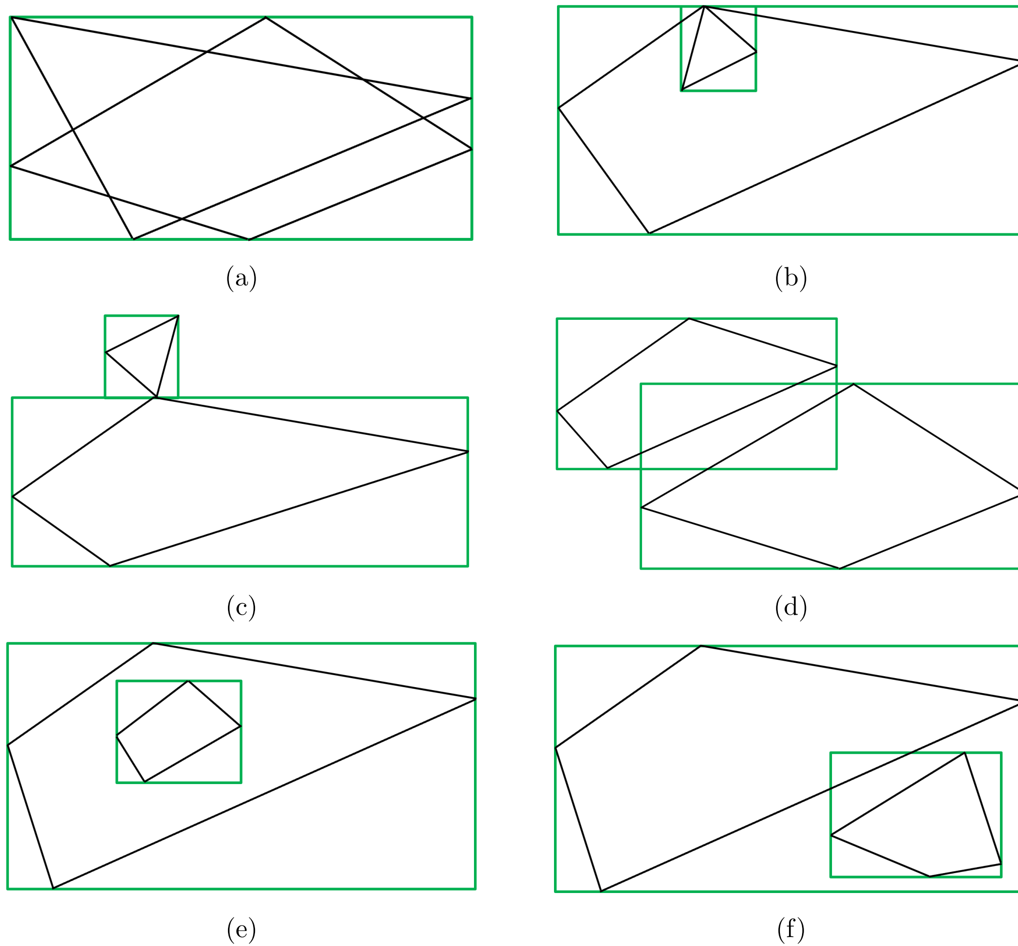


Figure 4.2: PNP cases: (a) Two polygons have line segment intersections, (b) and (c) Two polygons *touch* each other, (d) Two polygons' MBRs overlap but there is no actual intersection, (e) One polygon is inside another polygon but there is no line segment intersection, (f) The smaller polygon is not inside another polygon but the smaller MBR is inside another MBR.

S3 includes the tasks where two polygons do not have line segment intersection but one polygon may be inside another polygon. Then, we use the Striping Algorithm to preprocess data for the tasks in S1 which will be used in the

Stripe-based PNP function. Furthermore, we use the constant vertex PNP function only for the tasks in the S3 category. We discard the tasks in the S2 category. This helps in reducing a lot of PNP workload. For implementing filter and refinement steps, we use OpenMP and OpenACC to parallelize them on the CPU-GPU system.

Valid tasks for Stripe-based PNP function: If two polygons have line segment intersection(s) (e.g. Figure 4.2(a)), we store the pair for Striping algorithm and Stripe-based PNP Function. Moreover, we also store special cases as shown in Figure 4.2(b) and Figure 4.2(c) for further processing.

Valid tasks for constant vertex PNP function: If two polygons do not have any line segment intersection, we check their MBRs. If one MBR of a polygon is inside another MBR, we store this task for constant vertex PNP function. The reason is that the smaller polygon may be totally inside the larger polygon when they do not have a line segment intersection. In other words, all vertices of the smaller polygon may be inside the larger polygon. We also store which MBR includes another MBR because we only need to check whether the smaller polygon is inside the larger polygon. In addition, we do not need to check all vertices of the smaller polygon. It suffices to check a few vertices of smaller polygon whether they are inside or outside the larger polygon. Then, we know the smaller polygon is inside or outside the polygon. For illustration, Figure 4.2(e) and Figure 4.2(f) are two examples where we invoke the PNP function for only a few vertices. In the experiments, we consider the output of the PNP test for any five contiguous vertices to handle this special case.

Invalid tasks for PNP function: If two polygons intersect, there are two cases - a) there are line segment intersection(s) and b) there is no line segment intersection but one polygon is totally inside another polygon. Therefore, if two polygons do not have a line segment intersection and no polygon's MBR is inside another polygon's MBR, they do not intersect. As such, we will discard this task.

The reason is that if one polygon is inside another polygon, its MBR should be also inside another polygon's MBR. As shown in Figure 4.2(d), two polygons do not have any line segment intersection and no MBR of a polygon is inside another MBR, so they cannot intersect.

4.3.5 Striping Algorithm and Stripe-based PNP Function

We have used striping to speedup PNP tasks. Striping is a filter technique used to optimize the PNP function. Once the segments of the polygon are partitioned into stripes, the PNP test for a vertex needs to consider only the line segments contained in or crossing a stripe. This reduces the workload for PNP function.

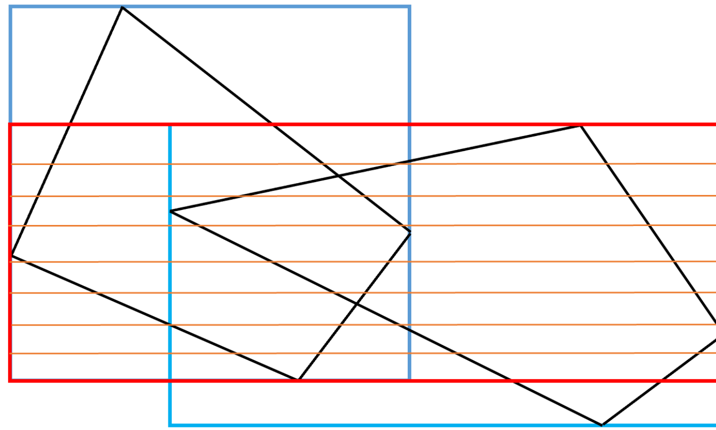


Figure 4.3: An example of striping for Stripe-based PNP function

The area divided into stripes is the red rectangle as shown in Figure 4.3. We divide the area occupied by the red rectangle into 8 cells. The Striping algorithm considers all the line segments belonging to a task and maps a line segment to the cells where there is an overlap. In case a line segment overlaps with two or more cells, the segment is replicated in those cells. This is carried out by comparing y co-ordinates of vertices of a line segment with the cell boundaries. For the Stripe-based PNP function, we need to check the vertices inside a cell only with

another polygon's line segments inside or across the same cell. Therefore, the vertices need to be compared only with those line segments which overlap with the same cell. For GIS datasets with large polygons, this can potentially reduce a lot of workload.

Multi-GPUs: After the geometries have been partitioned into multiple cells, parallel processing of PNP tests can be carried out over multiple GPUs. As shown in Figure 4.3, there is no dependency among those eight cells. In order to utilize four GPUs, we can assign two cells to each GPU in a round-robin fashion. In our experiments, we have leveraged multiple GPUs to distribute PNP-based computations.

4.4 Performance Evaluation

4.4.1 Performance of PNP Filters

Table 4.4: PNP based task reduction algorithm and striping effect on reducing workload of the PNP function for both the datasets and the reduction percentage

	Original workload	Current workload	Reduction percentage
Water	411,876,982,358	15,653,774,431	96.2%
Urban	6,453,160,088	291,816,678	95.48%

Table 4.4 shows PNP based task reduction algorithm and striping effect on reducing the workload of the PNP function for both the datasets and the reduction percentage. We can see that it reduced most of the workload. One reason is that we can discard some tasks where two polygons do not have any line segment intersection and the larger MBR does not contain the smaller MBR. Otherwise, there are only two types of tasks where we need to use the PNP function. One case is two polygons have line segment intersection(s) so there should be some vertices that are inside another polygon. Another case is when two polygons do not have

any line segment intersection but the larger MBR contains the smaller MBR so one polygon may be totally inside another polygon. Although we need to use the PNP function for these tasks, we have appropriate filters and refinement steps for them. For the first case, we use the Striping method to reduce the vertices, line segments, and workload. Striping can be effective when the CMBR of two polygons is large. For the second case, we check only a few vertices of two polygons to see which polygon is inside or outside another polygon based on our analysis. We only need to check a few vertices of only one polygon for the PNP function because the larger polygon cannot be inside the smaller polygon.

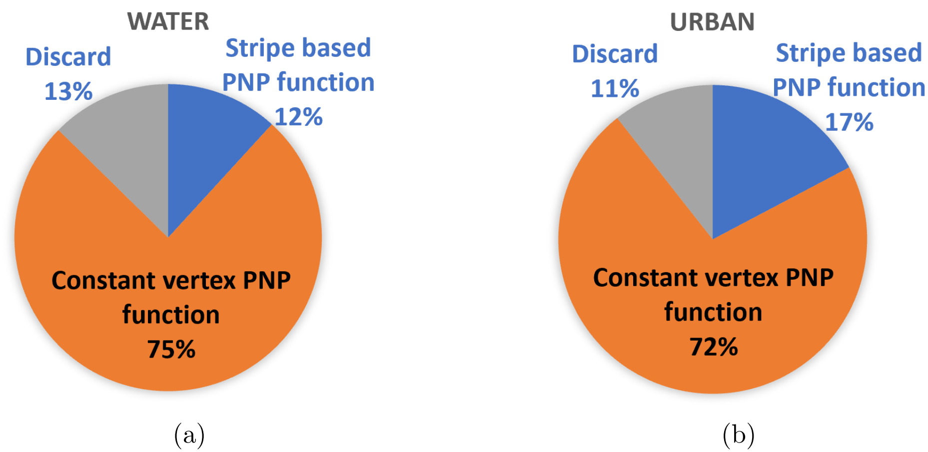


Figure 4.4: The percentage of different types of tasks after using PNP based task reduction algorithm

Figure 4.4 shows the percentage of different types of tasks after using PNP based task reduction algorithm. We can see that PNP based task reduction algorithm is very efficient. For the Water dataset, we need to check all vertices of polygons only for 12% of the tasks because the polygons of these tasks have line segment intersection(s). However, we can use the Striping method which is very helpful to reduce the workload of these tasks and discard some vertices which cannot be inside another polygon. Then, we use the Stripe-based PNP function. In addition, for 75% of the tasks, we can only use the constant vertex PNP function (5

vertices) of a polygon to determine whether they are inside or outside another polygon and then we know whether this polygon is inside or outside of another polygon according to our analysis. We can also discard 13% of the tasks and do not need to do PNP tests for these tasks. The PNP based task reduction algorithm also works well for the Urban dataset. We can discard 11% of the tasks and perform constant vertex PNP function for 72% of the tasks. Then, we use the Stripe-based PNP function for the remaining 17% of the tasks.

Table 4.5: Striping effect on reducing workload for the Stripe-based PNP function for both the datasets

	Original workload	Current workload	Reduction percentage
Water	156,443,271,335	5,301,138,126	96.61%
Urban	1,254,513,546	14,321,168	98.86%

Table 4.5 shows the Striping algorithm effect on reducing the workload of the Stripe-based PNP function for both the datasets. Stripe-based PNP function is applied to the tasks where two polygons have line segment intersection(s). We can see that it can reduce most of the workload for both datasets. The vertices inside a stripe need to be tested against the line segments only within the same stripe and crossing the stripe boundary, instead of all the line segments of a polygon. This leads to workload reduction. Even the area where we want to do striping is very large, we can still get benefits.

Table 4.6: PNP based Task Reduction Algorithm effect on reducing workload when a polygon MBR is inside another polygon MBR

	Original workload	Current workload	Reduction percentage
Water	152,287,577,854	10,352,636,305	93.2%
Urban	4,788,726,632	277,495,510	94.2%

In case when a polygon is inside another polygon, the MBR of a small

polygon is inside the MBR of a larger polygon. Our PNP-based task reduction algorithm detects these cases. So, we do not apply the quadratic time PNP tests in these cases. This results in workload reduction compared to the naive cases. Table 4.6 shows PNP based task reduction algorithm's effect on reducing workload of these cases. We only check five vertices of the smaller polygon to see whether these vertices are inside or outside of another polygon. Then, we determine whether the smaller polygon is inside or outside of the larger polygon.

4.4.2 Execution Time Results

Table 4.7: Execution time in seconds

Dataset	1 CPU thread and 1 GPU	32 CPU threads, 1 GPU for LSI and PNP	32 CPU threads, 1 GPU for LSI, multi-GPUs for PNP
Urban	1.39	0.35	0.30
Water	43.92	10.63	7.71

Table 4.7 shows the total run-times for two datasets. The first column's result is using 1 thread on CPU and 1 GPU without using our hierarchical filtering. For testing our system, we used one or more threads on CPU and one or multiple GPUs to see the difference. The second column's result is using 32 threads on CPU to preprocess data and 1 GPU for LSI and PNP function. The third column's result is using 32 threads on CPU to preprocess data, 1 GPU for LSI function, and multi-GPUs for PNP function. We can see that our system works well using multi-core CPU and multiple GPUs. Although we only use multi-GPUs for the PNP function, we can still get benefits, especially for the larger dataset.

Tables 4.8 and 4.9 show the details of execution time breakdown for the two datasets. We can see that our filters are very efficient and we can get benefit by using multi-GPUs. If the dataset is larger, we can get more benefit by using

multi-GPUs. For the Water dataset, the time taken by the R-tree filter on CPU is 2.27s. Therefore, the end-to-end time of the system is 9.98s.

Table 4.8: Execution time breakdown details for Water dataset. (NA means it is not applicable.)

Water	Sketch and LSI Function on GPU (s)	Pre-process for PNP on CPU(s)	PNP Function on GPU (s)	Final Time (s)
No filters	8.82	NA	35.1	43.92
One GPU for LSI and PNP	1.39	4.56	4.68	10.63
One GPU for LSI, multi-GPUs for PNP	1.39	4.55	1.77	7.71

Table 4.9: Execution time breakdown details for Urban dataset. (NA means it is not applicable.)

Urban	Sketch and LSI Function on GPU (s)	Pre-process for PNP on CPU(s)	PNP Function on GPU (s)	Final Time (s)
no filters	0.4	NA	0.99	1.39
One GPU for LSI and PNP	0.08	0.19	0.08	0.35
One GPU for LSI, multi-GPUs for PNP	0.07	0.19	0.04	0.30

4.4.3 CUDA information

For the Water dataset, the number of grids is 65535 and the number of blocks is 128 for *PolySketch* with the LSI function. We use multi-GPUs for the PNP function, so

we run PNP function A and B more times. However, the numbers of grids are still 65535 and the numbers of blocks are still 128.

For the Urban dataset, the number of grids is 28687 and the number of blocks is 128 for *PolySketch* with LSI function. As we mentioned before, we run PNP function A and PNP function B more times because of using multi-GPUs for the PNP function. For the PNP function B, the numbers of grids are 20629 and 58. The numbers of blocks are 128. For the PNP function A, the numbers of grids are 4956 and the numbers of blocks are 128.

4.5 Conclusion

We have developed a hierarchical PolySketch-based filter and refine system for GPUs and evaluated its performance using real-world datasets. Even though the system was implemented using compiler directives, the performance is very good. Spatial join on two large datasets can be performed in about 10 seconds. This is an order of magnitude better performance than the previous work where we did not leverage hierarchical filtering [14].

Chapter 5
Efficient Filters for Geometric Intersection Computations using GPU

5.1 Introduction

Geometric intersection algorithms are fundamental in spatial analysis in Geographic Information System (GIS). Applying high performance computing to perform geometric intersection on a 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 into the hierarchical filter and refinement (HiFiRe) system to solve the 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. As a result, we get on average 7.96X speedup compared to our prior version of the HiFiRe system.

Using PolySketch as one of the filters, a hierarchical filter and refinement system (HiFiRe) 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.

PolySketch filter uses sketch of a geometry represented by a set of contiguous MBRs (tiles) that approximate the geometry [68] 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 Chapter are as follows:

- PSCMBR Filter: Compared to the standard R-tree filter, the PSCMBR filter discards on average 76% of candidate pairs that 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 [68]. The workload after using tile-based PNP filter is on average 98% smaller than using constant vertex PNP filter [68].
- With the improved HiFiRe system equipped with new filters, we get on average 7.96X speedup compared to our prior version of the HiFiRe system. The processing rate of this new filter and refine system on GPU for reporting line segment intersection is 61 million segments/sec on average for real datasets.

5.2 Related Work

Common Minimum Bounding Rectangle: As shown in Figure 5.1(a), the blue and red rectangles are MBRs that each encloses a polygon and the green rectangle

is their CMBR.

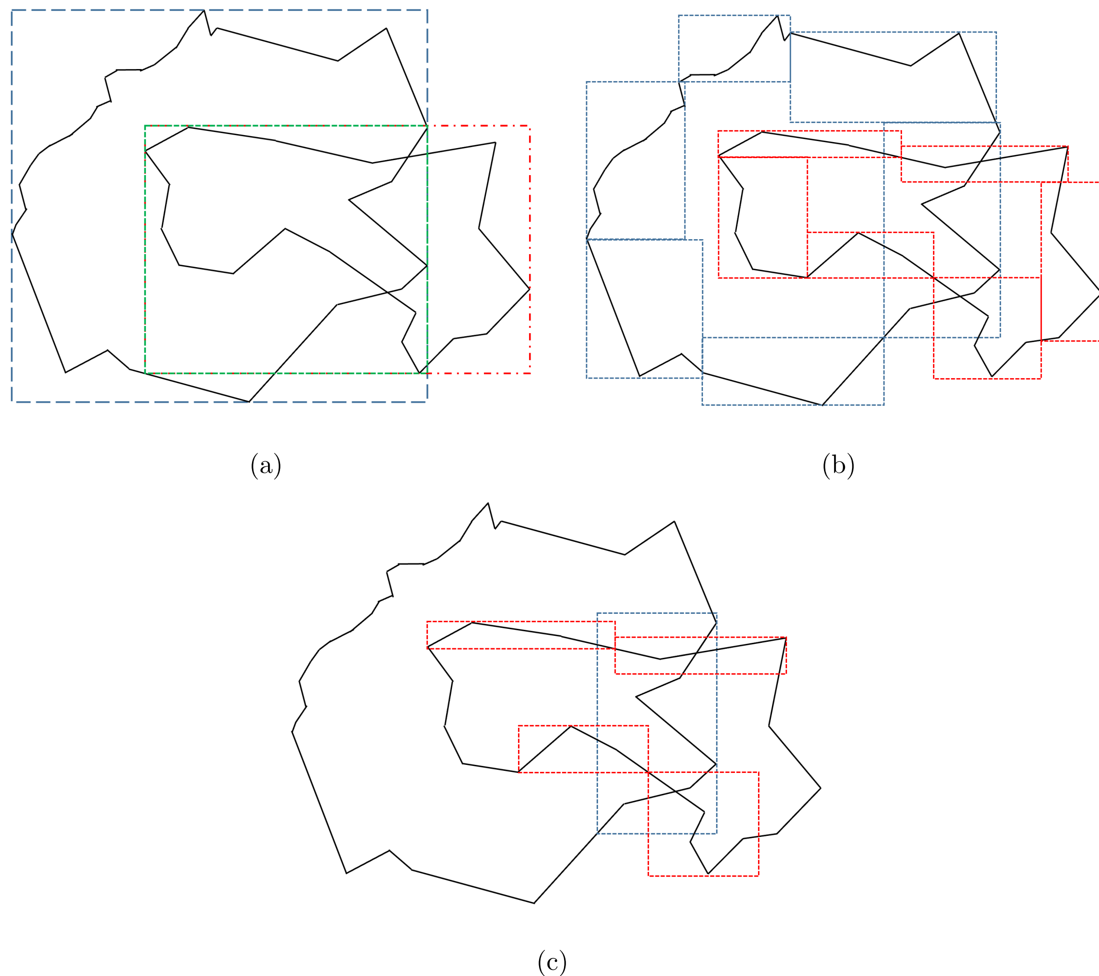


Figure 5.1: Two input polygons with (a) CMBR (green rectangle), (b) PolySketch showing the tiles and (c) only overlapping tiles after applying PolySketch filter.

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 for using a filter. However, the feature of CMBR makes it less effective if the CMBR is large as shown in Figure 5.1(a). Most line

segments are still in CMBR and cannot be ignored. In addition to the CMF method, to improve the efficiency of MBR, the paper [45] 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 [38, 39]. CMF is based on the common MBR area between two cross-layer polygonal MBRs. 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: Figure 5.1(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 5.1(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 the 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 [68]. 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 that 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 the 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 [68].

Another prior work in this area uses PixelBox technique which is pixel approximation of polygons for computation [25]. Geometries represented as 2D co-ordinates are converted to the raster format (pixels) to leverage image processing using a GPU [52]. Another work by Audet et al. [6] 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 [14].

5.3 PSCMBR Algorithm

5.3.1 Overview of PSCMBR Filter

As we discussed before, CMF and the 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 degrees 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 the 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.

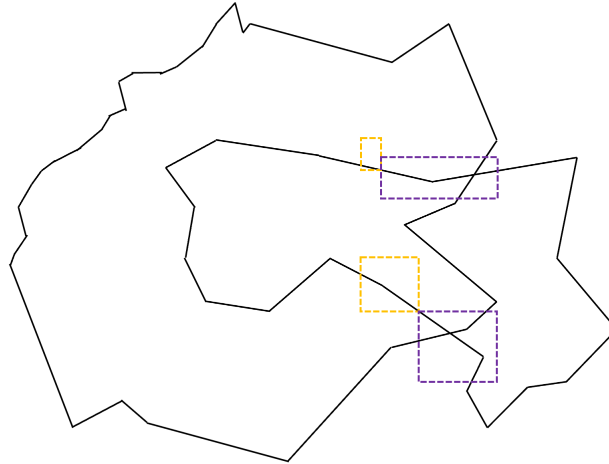


Figure 5.2: PSCMBR filter with four tile-CMBRs. Only the common area of overlap between the candidate tile pairs are retained (see Figure 5.1(c)).

We illustrate the PSCMBR filter in Figure 5.1. Figure 5.1(b) is the first step in using PSCMBR where we create sketches of polygons. There are four candidate tile pairs shown in Figure 5.1(c), where one tile of a polygon overlaps with four tiles of another polygon. Then, the PSCMBR filter calculates the CMBRs of a pair of tiles corresponding to the four candidate pairs. The four rectangles in Figure 5.2 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 the 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 a reduction in workload in the refinement phase.

Execution time model of intersection of two geometries using PSCMBR: We describe the workload in terms of tile-MBR intersections (two filters) and refinement using LSI. Table 5.1 defines the symbols used in modeling the

Table 5.1: 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

run-time 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.

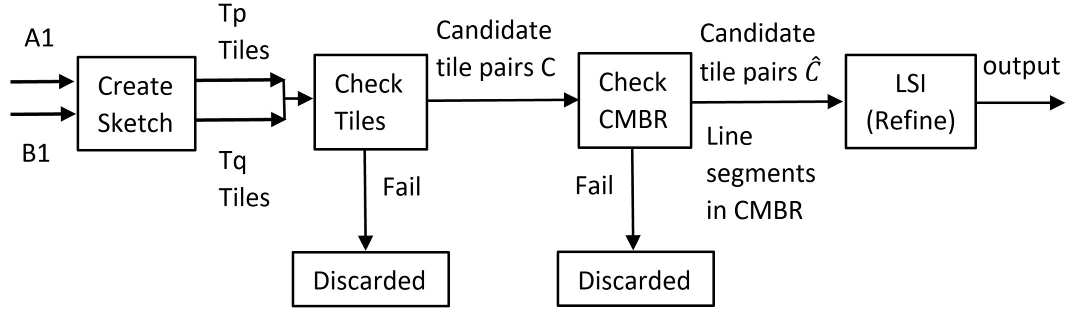


Figure 5.3: PSCMBR filter for reporting line segment intersections (LSI). Two polygons A1 and B1 are the input and the output is list of intersections.

Figure 5.3 shows the data-flow and control-flow in the PSCMBR filter and refine system. The input of PSCMBR is the candidate tasks. As shown in Figure 5.3, 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 run-time is given by the following equation:

$$\mathcal{T} = T_P \cdot T_Q \cdot \mathcal{T}_{MBR} + C \cdot \frac{P}{T_P} \cdot \frac{Q}{T_Q} \cdot \mathcal{T}_{LSI} \tag{5.1}$$

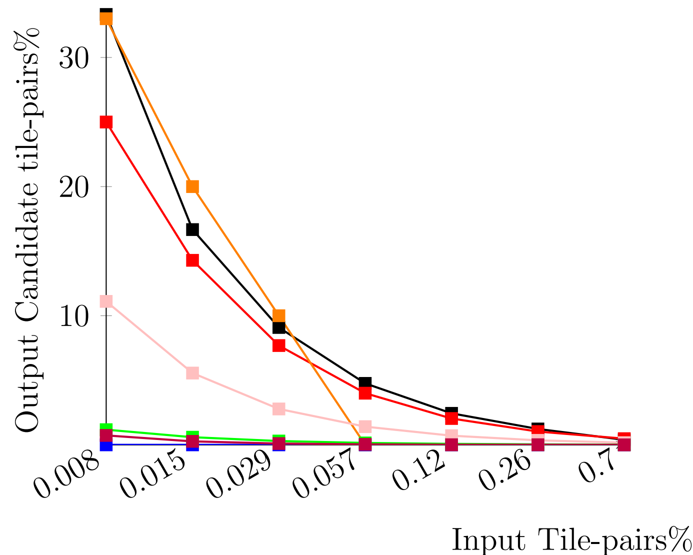


Figure 5.4: Effect of tile-size on the number of candidate tile-pairs. Input Tile-pairs%=$\frac{Tp*Tq}{P*Q}$*100. Candidate tile-pairs% = $\frac{C}{Tp*Tq}$*100. Each line denotes the output candidate tile-pairs% for a given input polygon-pair.

In Figure 5.4, we show the relationship between tile-size and the effectiveness of the PolySketch filter. The maximum number of tiles in a polygon is bounded by the number of line segments in the polygon 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 a very small (large) number of input tiles. In general, small tile sizes lead to better filter efficiency. However, a 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 run-time performance of the filter. Table 5.2 shows an example of the input polygon pair of Figure 5.4. 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.

Table 5.2: An example of input polygon pair in Figure 5.4

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%

We assume that the MBR overlap test is computationally cheaper than the line segment intersection for a pair of line segments, $\mathcal{T}_{MBR} < \mathcal{T}_{LSI}$. In addition, $\mathcal{T}_{CMBR} < \mathcal{T}_{LSI}$. From Equation 1, the LSI workload depends on the candidate tile-pairs. Overall performance depends on the delicate balance between the tile-MBR intersections and line segment intersections as shown in Equation 1.

Using PSCMBR, the run-time is given by the following equation:

$$\mathcal{T} = T_P \cdot T_Q \cdot \mathcal{T}_{MBR} + C \cdot \frac{P}{T_P} \cdot \frac{Q}{T_Q} \cdot \mathcal{T}_{CMBR} + \hat{C} \cdot \hat{P} \cdot \hat{Q} \cdot \mathcal{T}_{LSI} \tag{5.2}$$

5.3.2 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 overlapping the CMBR, instead of all line segments in a tile. Therefore, the refinement phase has fewer line segments to handle.

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 utilization. In addition, different GPU threads can be assigned to process different tiles of the same polygon, which increases the parallelization.

5.3.3 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 5.5 shows the pseudo-code of the PSCMBR filter applied to the 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.

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 5.3.4. 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 5.5 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 counts (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 a fewer number of tiles from the smaller polygon.

```
#define SIZE1 30
#define SIZE2 30
__global__ void PSCMBR(int numTileL1,int numTileL2,
    int tileSize1,int tileSize2,int *prefixSum1,
    int *prefixSum2,MBR *tileMBR1,MBR *tileMBR2,
    LineSegment *segment1,LineSegment *segment2,...){
  //We consider the larger polygon as the 1st polygon
  //numTileL1 is the number of tiles of the 1st polygon
  for (int j=threadIdx.x; j<numTileL1; j+=blockDim.x){
    for (int k=0; k<numTileL2; k++){
      if(tileMBR1[j] overlaps tileMBR2[k]){
        LineSegment subSegment1[SIZE1],subSegment2[SIZE2];
        int count1=0;
        int count2=0;
        //calculate the CMBR of two tiles
        MBR cmbr=getCMBR(tileMBR1[j],tileMBR2[k]);
        for(int jj=0; jj<tileSize1;jj++){
          if(segment1[prefixSum1[j]+jj] overlaps cmbr){
            storeSegment(subSegment1[count1],
                                segment1[prefixSum1[j]+jj]);
            count1++;
          }
        }
        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(subSegment1,subSegment2,count1,count2);
          }
        }
      }
    }
  }
}
```

Figure 5.5: CUDA Implementation of PSCMBR Filter

5.3.4 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.

5.4 Performance Evaluation

5.4.1 PSCMBR Filter Performance

Given two layers of polygons, the input to all the filters is the set of candidate polygon pairs obtained from R-tree query using a 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 the 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 used the symbols as described in Table 5.1.

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

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

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 $1 \cdot 3 \cdot 5 + 1 \cdot 10 \cdot 5 = 65$.

Table 5.3: Effect of different filters 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 5.4: Effect of different filters 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

Table 5.3, 5.4 and 5.5 show the performance of PSCMBR for three real datasets. Workload means the actual computational workload in the 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 the 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 5.5: Effect of different filters 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

According to Table 5.3, 5.4 and 5.5, 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 the refinement phase. In addition, the number of candidate tile pairs after using PSCMBR is on average 29.73% smaller than using PolySketch.

In the first step, PSCMBR discards more candidate tasks that do not need further refinement. In the second step, it discards more candidate tile pairs and reduces the false hits which do not need further refinement. According to Table 5.5, it reduces up to 47.6% of candidate tile pairs by using PSCMBR instead of PolySketch. For the run-time also, we can see PSCMBR works well. Compared to PolySketch, the 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 the Lake dataset is huge which leads to a higher overhead of copying the Lake data from CPU memory to the GPU memory for the PSCMBR filter. However, the data copy overhead is lower for CMF because we do pre-process 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 a little slower than the refinement time of LSI function after using CMF.

5.4.2 Performance of Using Different PSCMBR Tile-sizes

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 5.6 shows the performance of using different tile-size for the 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 5.6, for some rows, there are two numbers in 'tile-size' column. The first and second numbers are the tile sizes for

Table 5.6: 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

large and small polygons. In the experiments, if we use two tile-sizes for the polygons, we use a larger tile-size for the large polygons (with more than 400 vertices) and a smaller tile-size for the small polygons (with less than 400 vertices). We found that smaller tile-sizes perform better in our prior work [68]. We use different tile-sizes for the Water dataset to test the performance of PSCMBR.

According to Table 5.6, 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 a smaller tile-size can reduce more workload in the LSI function.

5.5 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 a PNP test for every point, which is an expensive operation. Therefore, we have designed a PNP filter based on PolySketch approximation. We assume that the PNP filter is invoked after the 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 [8, 7] 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.

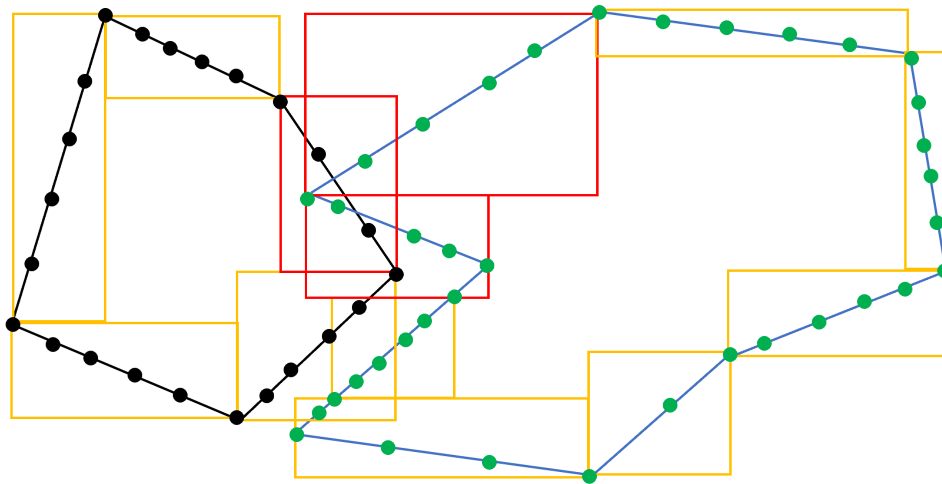


Figure 5.6: An example of *intersection tile* (red rectangle) and *no-intersection tile* (yellow rectangle)

5.5.1 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.

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 could be different from the status of the points that are after it if we traverse the points in clockwise order. We have observed that the number of intersection-points is far less than the input size of the overlapping polygons. Therefore, this limitation has very less impact on the performance.

As shown in Figure 5.6, 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.

Comparsion with our prior work: In our previous work [68], we had used space division (striping) to decrease the PNP workload where we 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 can be only contained completely inside another polygon or not. Overall, the PNP part was the bottleneck in earlier work [68, 6]. The for-loop in the ray-shooting algorithm was parallelized in [38] 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 [68].

5.5.2 Comparison of PNP Workload Reduction

Stripe-based PNP function: Stripe-based PNP function was proposed in the paper [68]. Using the 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

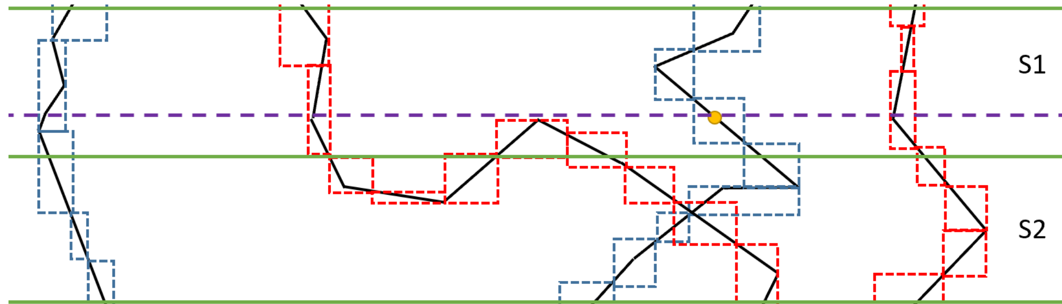


Figure 5.7: Illustration of PNP functions. Two polygonal chains extracted from input polygons is highlighted by red and blue colored tiles. The space is divided into two stripes S1 and S2. Dotted line is an imaginary ray parallel to X-axis and passing through the test point shown as yellow point.

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 the y-coordinates of line segments with the test point. Figure 5.7 shows how the Stripe-based PNP function works. In this example, the area is divided into two stripes S1 and S2 (the 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 5.7 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 the tile-based PNP function, it should be compared with the line segments within only two tiles (considering the y coordinate of the test vertex).

5.5.3 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, the 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, the PolySketch-based PNP function is used for tasks where two polygons have line segment intersection points. In addition, the tile-based PNP function can be also used here to reduce the workload. Therefore, we apply these two filters together in the same kernel.

5.6 Performance Evaluation

5.6.1 System Performance with PNP Filters

Table 5.7: System run-time (does not include R-tree time)

	HiFiRe run-time(s) [68]	New HiFiRe run-time(s)
Urban	0.35	0.055
Water	10.63	1.109

Table 5.7 shows the system run-time results. To be fair, we compare them with the results of using one GPU. We can see the performance of the system is much improved. The new system gets 6.36X and 9.56X speedup compared to the

HiFiRe system for the Urban and Water data sets. One reason is that we use GPU to pre-process data for the 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 the PNP test because we can make full use of the tiles used in the 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.

5.6.2 Filters with PNP Test Workload

Table 5.8: 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

To show the efficiency, we compare the PolySketch-based PNP function with the Stripe-based PNP function (using 8 stripes). We also compare tile-based PNP function with constant vertex PNP function. For the workload of the 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 $2 \cdot P \cdot Q$ for every task. In addition, the total workload for the PNP test is the summation of the workload of individual tasks. Table 5.8 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 is different, we also update the workload of Stripe-based PNP [68].

We can see that the workload is still much reduced by using the PolySketch-based PNP function even compared with the Stripe-based PNP test. For Urban and Water, it reduces 41.2% and 79.2% of the workload of Stripe-based PNP function.

PolySketch-based PNP function classifies the tiles into two categories, namely, *intersection tile* and *no intersection tile*. Figure 5.8 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 the PNP filter.

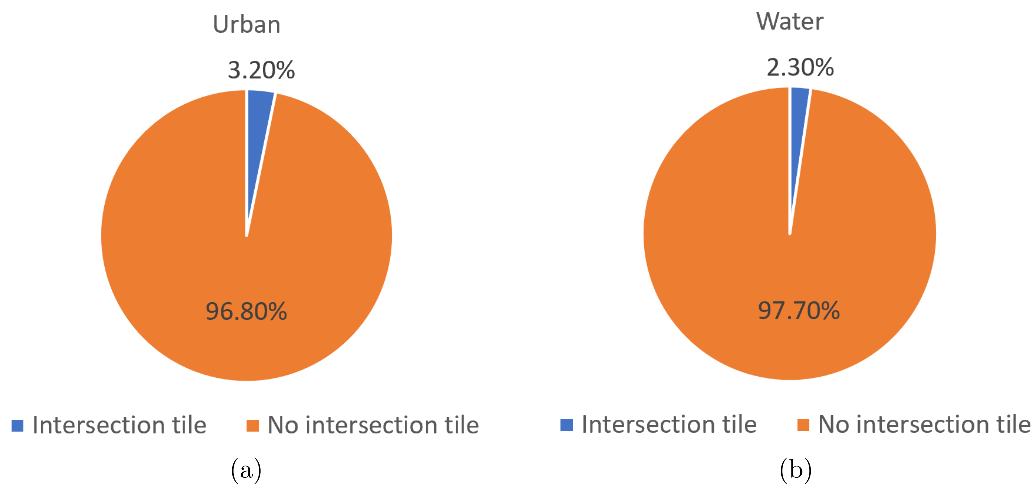


Figure 5.8: Percentage of *Intersection tile* and *no intersection tile* for tasks where two polygons have line segment intersections

For the *intersection tile*, we can see the percentage is 3.2% and 2.3%. Although we have to do the PNP test for all vertices within intersection tiles, the total number of such tiles is not large. In addition, the 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 5.9, we can see tile-based PNP function can also reduce

Table 5.9: The workload in PNP test for the tasks where one polygon may be totally inside another polygon

	Constant vertex PNP workload [68]	Tile-based PNP workload
Urban	277,495,510	3,452,066
Water	10,352,636,305	145,693,382

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 the test vertex by only considering y-coordinate.

5.6.3 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 the PSCMBR HiFiRe system, we define the processing rate in terms of millions of input line segments/second:

Processing rate = Input line segments/Overlay processing time.

For the 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 85,044,509. The processing rate on GPU is 77 million segments/sec. The processing rate of the end-to-end system is 25 million segments/sec. For the 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 on GPU is 45 million segments/sec. The processing rate of the end-to-end system is 21 million segments/sec.

5.7 Conclusion

We have developed new filters used in the filter and refine technique and demonstrated the benefits of 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 the HiFiRe system.

Chapter 6
Adaptive Filter for Geometric Intersection with Rectangle Intersection Computations using GPU

6.1 Introduction

In the previous chapters, we have introduced *PolySketch* filter which is a filter for geometry-geometry intersection algorithms. In this chapter, we introduce a new two-step geospatial filter called *PolySketch++*. It refines the sketch dynamically to decrease the overall computations in the filter and refine phases. This improves the basic *PolySketch* filter by making the tile size adaptive. The adaptive nature makes the filter general and allows it to handle very large polygons efficiently as well. We have implemented such filters on GPU by using CUDA instead of OpenACC. CUDA dynamic parallelism was also applied to the new adaptive filter called *PolySketch++*. With *PolySketch++*, we have demonstrated performance gain by using GPU dynamic parallelism in geometry-geometry intersection operation.

Geospatial computations like spatial join and map overlay exhibit irregular workload because of the non-uniform spatial distribution and variable size of geometries [15]. The workload is not available statically which motivates dynamic approaches. Workload calculation requires computations with approximate representations called MBR. When the data is spatially partitioned among grid-cells, the workload across individual cells varies. When the data is not partitioned, then workload across candidate pairs varies.

Spatial hierarchical data structures like Quadtree adapt to the spatial distribution of the 2D data; regions that are dense are subdivided into four regions recursively. *PolySketch* filter represents the boundary of a geometry in terms of a collection of contiguous tiles where each tile is a subset of the vertices and its associated MBR. Our new adaptive filter is different because the filter adapts to the

workload in polygon/polyline intersection. As such our new filter is workload-aware vs other spatial techniques that are density-aware. This adaptive technique is a good fit for the output-sensitive geometric intersection computations because the time complexity depends on the size of the output (number of segment intersections) produced by the algorithm.

PolySketch uses fixed tile size which is determined a priori. In this chapter, we present adaptive *PolySketch* which we refer to as *PolySketch*++. *PolySketch*++ is an adaptive and dynamic filter where the tile size adapts to workload distribution inherent in the intersection of two geometries. We use CUDA dynamic parallelism for implementation where a kernel can launch another kernel without CPU involvement.

The contribution is as follow

- *PolySketch*++: An adaptive spatial filter for polygon and polyline intersection using CUDA dynamic parallelism. For huge polygons, we get on average 4.23X speedup compared to *PolySketch* filter. It reduces on average 92.5% of workload in the filter phase compared to *PolySketch* and also reduces workload in refine phase.
- LMBR: A spatial filter for rectangle intersection using CUDA. In the HiFiRe system, it works better than using R-tree to create candidate tasks.

6.2 Related work

Adaptive space partitioning for load balancing in spatial join using two collections of geometries has been studied in [69]. *PolySketch*++ can be leveraged for a pair of geometries that is one of the candidates in spatial join.

From Kepler architecture, NVidia GPUs support dynamic parallelism that is a CUDA technique where kernels can launch kernels. This allows a thread to launch a new grid of threads to execute another kernel. It can be used for recursive

algorithms, irregular grid structures, and others. It can reduce the need to transfer execution control and data between device and host. It can be useful for the problems where need nested parallelism. For example, using recursion, hierarchical data structures which can be adaptive grids, and others.

Dynamic parallelism can benefit CUDA kernels where the workload is irregular across the tasks. However, to be practical the benefits must outweigh the cost of launching additional child kernels from the parent kernel. The overhead associated with kernel launch is significant because the state of thousands of threads needs to be saved. Dynamic parallelism (DP) has been studied earlier on GPUs using a compiler directive-based approach for irregular and nested loops [55, 70, 54]. The overheads associated with dynamic parallelism have been presented in [71, 72]. Spatial computations have not been studied earlier with DP.

6.3 PolySketch++ Algorithm

6.3.1 Overview of Adaptive PolySketch

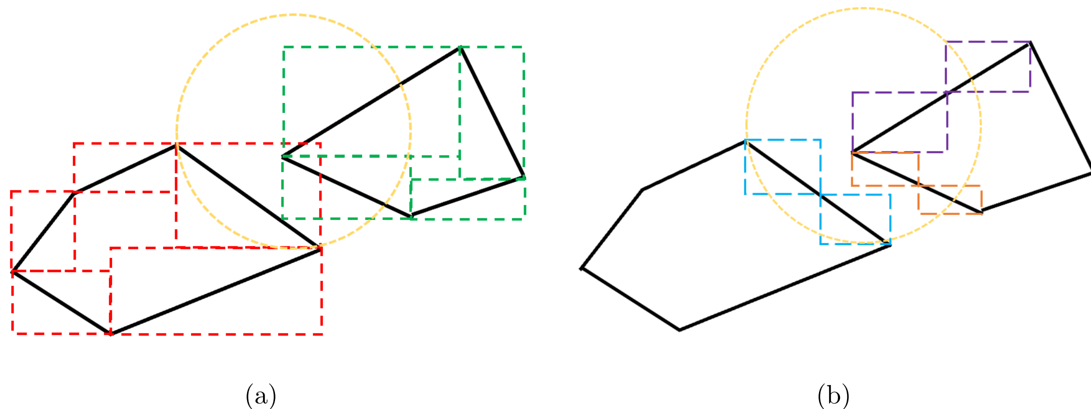


Figure 6.1: PolySketch++ filter example (a) Overlapping tiles highlighted by a circle. It shows a red tile overlapping with two green tiles, and (b) The overlapping tiles are further subdivided. Further refinement is not required.

PolySketch has been described in the previous chapters. The sketch is made

up of tiles. Once a tile-size is fixed, then an MBR is calculated which is known as tile-MBR. The disadvantage in using *PolySketch* is that it is difficult to find an optimal granularity for tile-size statically. As such, the performance of *PolySketch* filter is dependent on the appropriate selection of tile-size which has to be manually tuned.

The dynamic tiling approach that we introduce here subdivides only the candidate tiles. The main idea in *PolySketch++* is to perform filtering in two phases using CUDA dynamic parallelism. In the first phase, we start with a coarse-grained tile size in the parent kernel which discovers work and the child kernel makes the tiling fine-grained. As opposed to the static approach, where only a block of threads was assigned for a pair of geometries, in the dynamic approach, the number of thread blocks is proportional to the workload in geometric intersection. Moreover, compared to the static approach, we get better load balancing at the cost of overhead for dynamic kernel launches.

Figure 6.1 shows how the adaptive *PolySketch* filter works. At first, we use a large tile size called the parent tile size and check which tile's MBR overlaps with others. A pair of overlapping tiles is called a candidate tile-pair, e.g., a red and a green tile. Going forward, we discard the tiles whose MBR does not overlap with others. Then, for the candidate tile-pairs, we use a smaller tile-size called the child tile size to divide the remaining tiles. We can repeat the filter using smaller tiles and find the new candidate pairs. Finally, the refinement phase is carried out with the remaining candidate pairs. This filter-and-refine approach potentially reduces the number of expensive line segment intersection tests (LSI). As shown in Figure 6.1(a), there are two polygons with their respective tiles. A red tile overlaps with two green tiles. So, there are only two candidate tile-pairs that are used; the rest of them are discarded going forward. The candidate tile-pairs are subdivided to produce smaller tiles. The subdivision is implemented by reducing the tile-size by

half and recalculating the MBR for the line segments in the new tile. As shown in Figure 6.1(b), there is no tile MBR overlap pair and we do not need to invoke the LSI function. This is an example to show how adaptive *PolySketch* works.

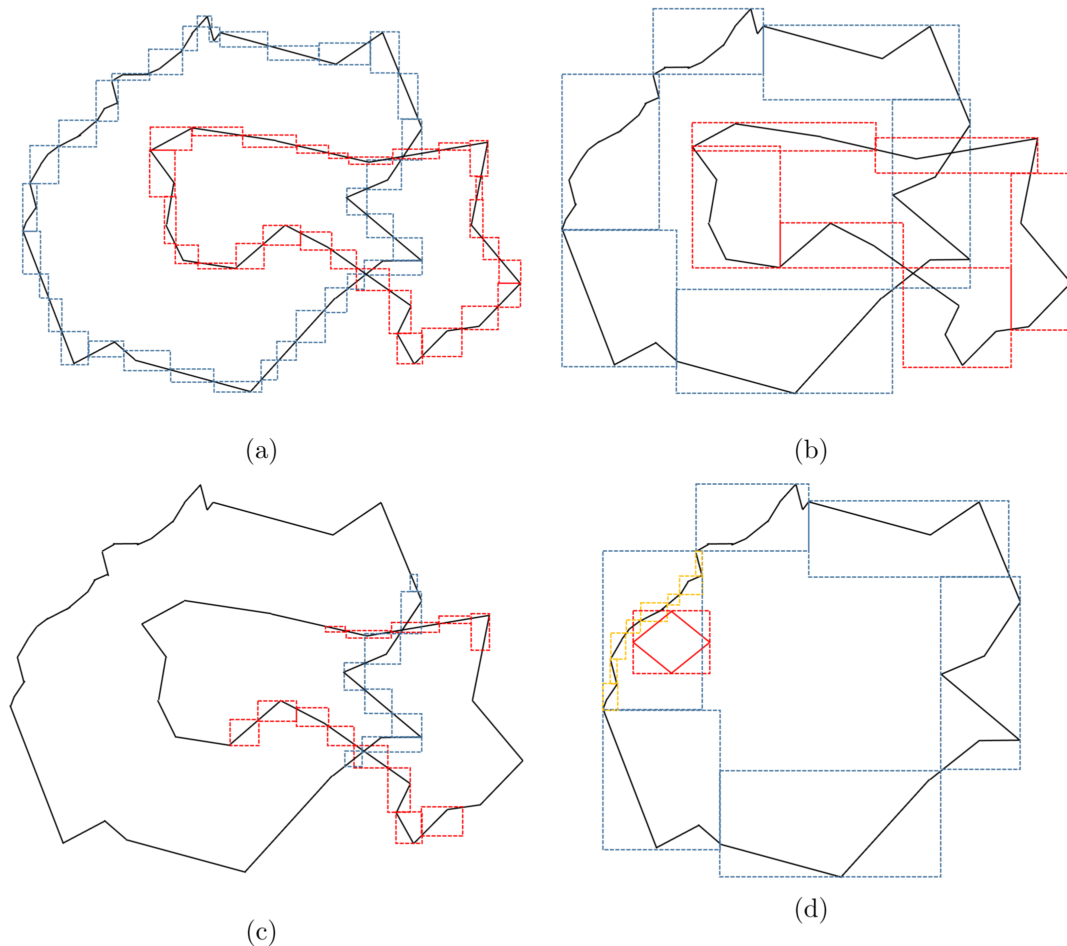


Figure 6.2: Static filter shown in (a). Two-step adaptive filter shown in (b) and (c). (b) shows filter using coarse-grained tile size and (c) shows fine-grained tile size for overlapping tiles only; (d) An example where two polygons have large difference in sizes.

Now we want to show the main difference between static tiling vs dynamic tiling for implementing a polygon intersection filter. We use Figure 6.2 for illustration where the input is two polygons and the output requires finding all the line segment intersections. The first three sub-figures are about two polygons of

similar size. The last sub-figure is for a large and a small polygon. As shown in Figure 6.2(a), we create polysketch for the two polygons and use the sketches for filtering out the tiles that have overlap. In the worst case scenario, if we have T_P and T_Q number of tiles in the two polygons, then $T_P \cdot T_Q$ comparisons are required. As we can see that very few tiles overlap from the two polygons.

Another approach is shown in the next two figures, namely, Figure 6.2(b) and 6.2(c), where a larger tile-size is selected to create the sketch compared to Figure 6.2(a). Then a smaller tile-size is selected for re-creating the sketch for the candidate tile-pairs. The benefit of this approach is that we need fewer tile-MBR comparisons in the filter phase in general. As shown in the illustration, even with fewer tiles, many tiles that do not have overlap can be safely ignored. Moreover, the refinement workload is same in both approaches. This approach is dynamic and adaptive.

When we combine two real-world maps, we see that one polygon from *Lakes* can overlap with many other smaller polygons from *Counties*. Figure 6.2(d) shows such an example. We can see that setting the tile-size adaptively for the larger polygon requires fewer MBR comparisons than using a static tile-size. The performance of the filter is data-dependent. The number of tiles chosen in the first and second filters affects the run-time of the algorithm. Next, we will show the run-time formula of our filter and refine method.

Table 6.1: Symbol Table

Symbol	*Definition*
$\mathcal{T}_{MBR}$	Time for checking if two MBRs overlap
$\mathcal{T}_{LSI}$	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 PolySketch++
$\hat{P}, \hat{Q}$	Number of line segments in new tiles

Execution time model of Intersection of two geometries using PolySketch and PolySketch++: Table 6.1 shows the symbols used in deriving the run-time formula of intersection of two geometries (polygons). T_P and T_Q are numbers of tiles of two polygons with P and Q line segments respectively. $\frac{P}{T_P}$ and $\frac{Q}{T_Q}$ are numbers of line segments in a tile (tile-size) of the respective geometries.

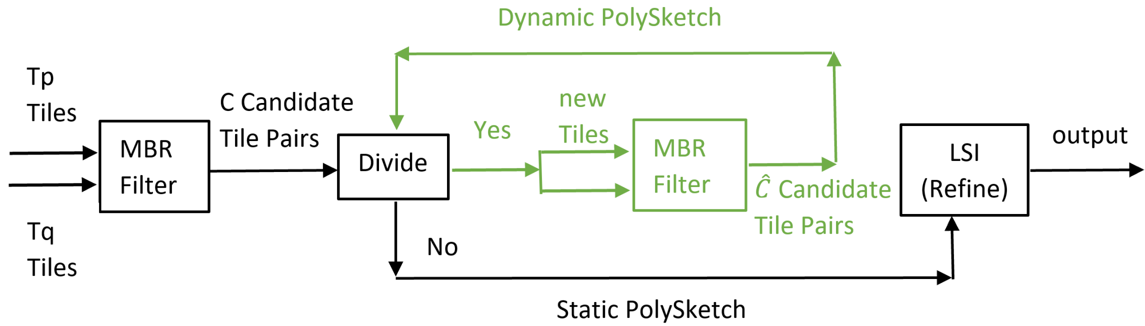


Figure 6.3: PolySketch++ filter for speeding up line segment intersections (LSI). The filter is dynamic and recursive. The Divide component produces more tiles (by factor of a and b) by subdivision of input tiles.

Figure 6.3 shows the data-flow and control-flow in our dynamic filter and refine system. It also shows the difference between *PolySketch* and *PolySketch++* filter by using different colors. The output of the first MBR filter is a collection of candidate tile-pairs that is C. We have $C <= T_P \cdot T_Q$ since not all tiles have spatial overlap.

We assume that the MBR overlap test is computationally cheaper than the line segment intersection for a pair of line segments, $\mathcal{T}_{MBR} << \mathcal{T}_{LSI}$. Using *PolySketch*, the run-time is given by the following formula

$$\mathcal{T} = T_P \cdot T_Q \cdot \mathcal{T}_{MBR} + C \cdot \frac{P}{T_P} \cdot \frac{Q}{T_Q} \cdot \mathcal{T}_{LSI} \tag{6.1}$$

Using adaptive *PolySketch*, the run-time is given by the following formula:

$$\mathcal{T} = T_P \cdot T_Q \cdot \mathcal{T}_{MBR} + C \cdot \mathcal{T}_{MBR} + \hat{C} \cdot \hat{P} \cdot \hat{Q} \cdot \mathcal{T}_{LSI} \tag{6.2}$$

In the run-time formula, we have shown two filters and one refinement. Conceptually, we can have more than two filters in a recursive fashion. This design is shown in Figure 6.3 by connecting the output of the second filter to the *Divide* component. Given a candidate tile pair, the *Divide* component can divide 1) tiles of both geometries or 2) tiles of only one geometry (larger among the two geometry). The goal of the *Divide* component is to stop the recursion by comparing the benefit of the workload reduction in a step with the overhead of recursion. The workload is captured by C, $\hat{C}$, number of tiles and tile sizes. When *Divide* component returns "No", then the static workflow is followed by executing the final refinement step.

The number of recursive steps that can be efficiently executed is data dependent. In multi-core CPU, the overhead of recursion is relatively lower than in GPU. In GPU, the state of thousands of threads needs to be stored when a child kernel is launched from the parent kernel, which makes the overhead of the child kernel launch higher. For GPU algorithm implementation, we used only one child kernel for each overlapping polygon pair.

6.3.2 Advantages of Adaptive PolySketch

As we showed in the figures earlier, using a two-step filter can be advantageous. Compared to using a very small static tile size, using a coarse-grained tile size in the first filter can decrease the number of tiles examined at the beginning. It means that we get fewer tiles to be checked and still can discard the non-overlapping areas of polygons. Then, for the areas where two polygons may intersect (overlap), using the second filter with a smaller tile size can still discard some tiles and even reduce workload in the refinement phase. Moreover, we do not need to run a number of experiments to pick a good tile size. Even if the parent tile size is not optimal, the

child tile size can be chosen appropriately to balance the workload.

Another advantage of adaptive *PolySketch* is in the space utilization because it makes good use of limited GPU memory. For huge polygons, if we use static *PolySketch*, there can be potentially a huge number of tiles to get a good filter efficiency to handle an arbitrary input. We need to store the information for those tiles, such as tiles' MBR. If we have a lot of tasks with huge polygons, the memory requirement could become a limiting factor. The feature of the adaptive tile size solves this problem.

6.3.3 The Implementation of Adaptive PolySketch and CUDA Dynamic Parallelism

Implementing Adaptive *PolySketch* filter on GPU is not straightforward. There are some factors that can affect its performance: how to make full use of the GPU resource, how to store the results after using the parent tile size, how to assign threads to the work, and so on. For different tasks, the numbers of tiles in the two polygons are different. For different tiles of one polygon, the numbers of candidate tile-pairs are also different. It is not space-efficient to store all these intermediate results on GPU with limited global memory and use them effectively. Therefore, we use CUDA dynamic parallelism which allows users to launch kernels from threads running on the device. For example, we can call a 'child' kernel within a 'parent' kernel. This gives us more choices and flexibility to improve and implement an algorithm. It can be used to process different levels of details for different inputs or recursive algorithms. Typically, this can decrease the data movement, store fewer data and make the algorithm more efficient. However, the kernel launch overhead, race conditions, optimal selection of threads and blocks need to be considered.

Figure 6.4 shows the pseudo-code of *PolySketch++* filter applied to LSI function using CUDA dynamic parallelism. A task is a pair of overlapping polygons

where we apply the filter and the algorithm handles a large number of such tasks. Although we have two polygons in a task, we only use the adaptive tile size for the larger polygon to reduce the overhead of calling child kernels and we do only 2 kernels. The 1st polygon in the algorithm means the polygon that has more tiles in a task.

```
1   //create different blocks to different tasks
2   //t1 and t2 are the number of threads used in kernels
3   PolySketch_Parent <<<tasks,t1>>>(TileL1, TileL2,
4                              TileMBR1, TileMBR2,...);
5  __global__ void PolySketch_Parent(int *TileL1,
6        int *TileL2, long *TileMBR1,long *TileMBR2,...){
7   int i = blockIdx.x;
8   //We consider the larger polygon as the 1st polygon
9   //TileL1[i] is the number of tiles of the 1st polygon
10  for (int j=threadIdx.x; j<TileL1[i]; j+=blockDim.x){
11    for (int k=0; k<TileL2[i]; k++){
12      if(TileMBR1[j] overlaps TileMBR2[k]){
13        cudaStream_t s;
14        cudaStreamCreateWithFlags(&s,
15                                cudaStreamNonBlocking);
16        PolySketch_Child <<<1, t2, s>>> (TileMBR1,
17                                TileMBR2, j, k,...);
18      }
19    }
20  }
21 }
22
23 __global__ void PolySketch_Child(long *TileMBR1,
24                     long *TileMBR2, int j, int k,...){
25  //newTile is the number of small tiles of
26  //the previous Tile(j)
27  for (int jj=threadIdx.x; jj<newTile; jj+=blockDim.x){
28    //Calculate the new small Tile[jj]'s MBR
29    //called newTileMBR
30    getNewTileMBR(TileMBR1[j],newTileMBR);
31    if(newTileMBR overlaps TileMBR2[k]){
32      //Calculate and store the line segment intersections
33      LSI(Tile1[jj],Tile2[k]);
34    }
35  }
36 }
```

Figure 6.4: CUDA Implementation of PolySketch++ Filter

Generally, grids launched within a thread block are executed sequentially. In

other words, a grid can start execution only after the previous grid finished execution even if these grids are launched by different threads within the same block. However, we prefer to let these grids launched by different threads start executing at the same time. Therefore, we use CUDA streams to implement this. Kernels launched in different streams will be executed concurrently. The streams are created by cudaStreamCreateWithFlags() API with cudaStreamNonBlocking flag.

In CUDA, every kernel will be executed on Streaming Multiprocessors that will try to divide the threads into Warps whose size is 32 threads. Since a 'child kernel' will create a new grid, we should divide the filter computations into at least 32 threads to make full use of the resources. Therefore, if the child tile size is very small (such as 4 line segments), the parent tile size should be at least 128 line segments. It means the large tile can be divided into 32 small tiles and 32 threads are used for calculations. If the parent tile size is below 128 line segments, we can not make full use of the threads in a warp.

6.4 Experimental Setup and Results

6.4.1 Data Sets

Table 6.2: Three real data sets used in our experiments for PolySketch++

Label	Dataset	Polygons	Segments
Classic	*Classic*	1	101K
		1	72K
Water2	*The subsets of Water*	1,172	88K
		1	86K
Ocean	*ne_10m_ocean*	1	446K
	continent	8	181K

We have used three real data sets to evaluate *PolySketch++*: (1) Classic, (2) Ocean, and (3) Water2. Water2 is a small subset of the Water dataset. Some

details are shown in Table 6.2. The Ocean dataset is from http://www.naturalearthdata.com and http://resources.arcgis.com. The Classic dataset is from https://rogue-modron.blogspot.com/2011/04/polygon-clipping-wrapper-benchmark.html.

6.4.2 Adaptive PolySketch Run-Time

We apply *PolySketch++* in the LSI function using Classic, Ocean, and Water2. First, we calculate the total number of tiles in polygons, tiles' MBR, and copy data between CPU and GPU. For Classic, Ocean, and Water2, it takes 1ms, 7ms, and 250ms. While comparing the performance, we did not add these times to the tables described next. We have implemented both *PolySketch* and *PolySketch++* using CUDA on GPU. The only player affecting performance is adaptive filtering.

Table 6.3 is about the performance of using *PolySketch++* and *PolySketch*. For *PolySketch*, we can directly get better performance by using more threads and we put the best result after we tried different static tile sizes. We also put the best result of *PolySketch++*. For Classic, the static tile size is 24 line segments and the number of threads in one block is 512. The parent tile size is 512 and the child size is 4. The number of threads is 256 in the parent kernel and 128 in the child kernel. For Water2 and Ocean, the static tile size is 15 (for large polygons) and 5 (for small polygons). The number of threads in one block is 512. For adaptive tile size, we varied the tile sizes from 128 to 3072. We varied the threads from 32 to 512 as shown in Figure 6.5.

Table 6.3: The run-time using static and adaptive tile size.

	PolySketch (static)	PolySketch++ (adaptive)
Classic(ms)	95.2	9.6
Water2(ms)	168.5	140.9
Ocean(ms)	593.4	373

Table 6.3 shows that *PolySketch++* works better than *PolySketch*. For Classic, Water2, and Ocean, we get 9.92X, 1.19X, and 1.59X speedup compared to *PolySketch*. Compared with *PolySketch*, the main advantage of *PolySketch++* is that we can discard a similar number of tiles but checking the fewer number of tiles and we can still reduce more LSI workload.

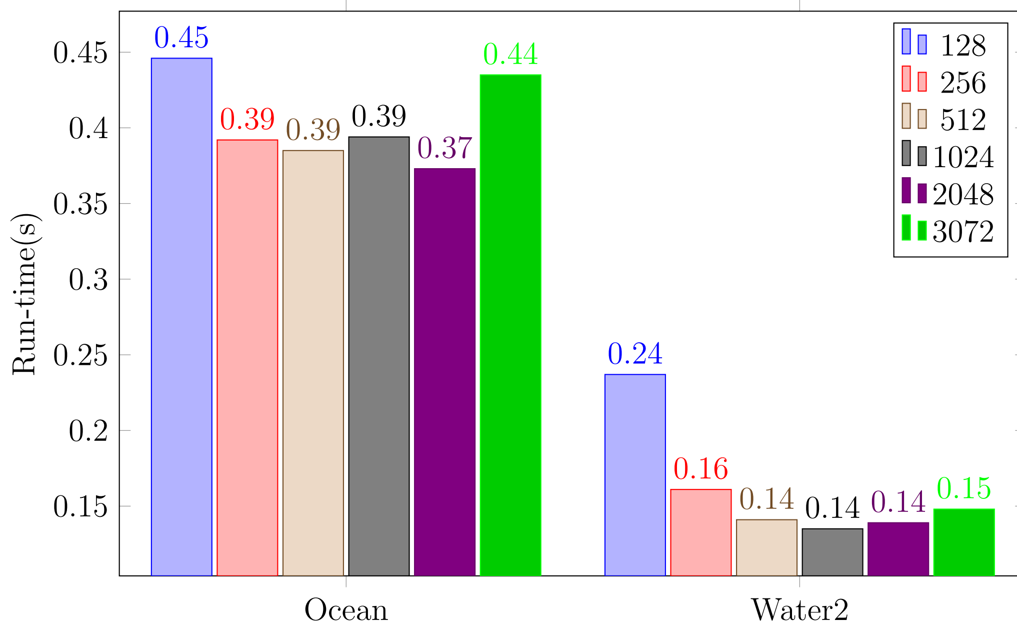


Figure 6.5: Execution-time variation using different parent tile-sizes (128 to 3072). Number of threads are adjusted based on tile-sizes (maximum number of threads are assigned to parent and child kernels based on the workload).

6.4.3 Effect of Varying Tile Sizes and Threads

Here, we want to study the performance of using different tile-sizes with different numbers of threads assigned to parent and child kernels. Figure 6.5 shows the results of setting the parent tile size as 256, 512, 1024 or 2048 are similar if we assign the maximum number of threads to the parent and child kernels based on the workload. For example, the maximum number of threads assigned to the child kernel is 64 if the parent tile can be divided into only 64 tiles. Otherwise, there is not enough

work to be assigned if we use 256 threads, so 192 threads will not get enough work to do. We have more flexibility in choosing a parent tile size because the child tile size can still improve the performance even if the parent tile size is not the best one.

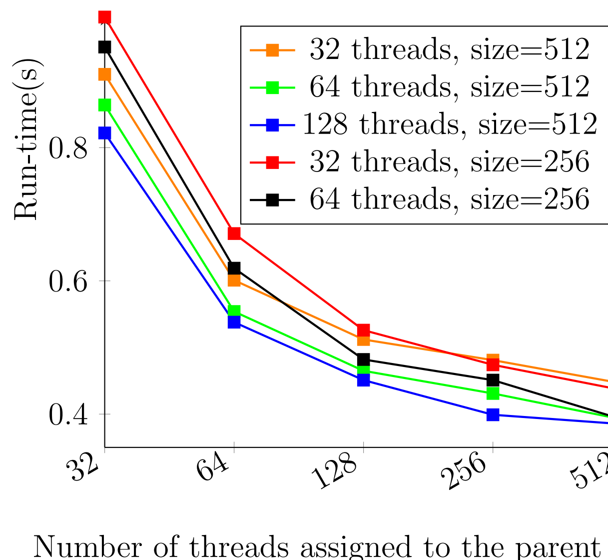


Figure 6.6: Effect of using different parent tile sizes and different number of threads assigned to the parent and child kernels

Figure 6.6 shows the run-time using Ocean data. We can see that using more threads in the parent or child kernel can directly improve the performance when we use the same parent tile size. If we use the same number of threads in the parent and child kernels and use different parent tile sizes, the difference in performance is large when we use a small number of threads. The difference is small if we use a large number of threads. Although we used two different parent tile sizes, the results are similar when we use the maximum number of threads to the parent and child kernels. Therefore, we should divide the polygon into some tiles which can use more threads, such as 256 or more threads. Then, one large tile should be also divided into more tiles to be assigned to threads, such as 64 or more threads. Generally, we set the child tile size as 4 and do not change it. According to our

experiments, a larger parent tile size is recommended if we can use 256 threads or more in the parent kernel. Based on the results, 512 and 1024 line segments are good choices for the parent tile size.

6.4.4 Adaptive PolySketch Workload

The workload is divided between the filter phase and the refine phase. The filter phase workload is referred to as tile workload and the refine phase workload is referred to as LSI workload. By using the adaptive or static tile size, we can reduce the number of line segments which should be compared with other line segments. Therefore, we call the LSI function (refine) workload after the *PolySketch* or *PolySketch++* filter as 'static LSI workload' and 'adaptive LSI workload'.

By using adaptive *PolySketch*, we can reduce the filter time. To show the benefit, we calculate the workload which is the summation of how many times one tile is compared with other tiles for every tile of every task. We call this total workload as 'Tile workload'. For example, A has g tiles and B has h tiles. The tile workload is $g \cdot h$ for every task. The total tile workload is the summation of the workload of every task. In addition, the results are different if we use different adaptive tile sizes.

Table 6.4: The LSI function workload and tile workload

	Static LSI workload	Adaptive LSI workload	Static tile workload	Adaptive tile workload
Classic	449,722	336,684	37,707,648	1,258,258
Water2	38,656,534	36,428,698	80,207,239	6,736,435
Ocean	69,560,256	33,643,788	140,513,568	14,987,132

Table 6.4 shows the LSI function workload and tile workload of different data sets. We compare the results by using *PolySketch* or *PolySketch++*. We set the parent tile size as 512 line segments and the child tile size as 4 line segments. We

can see that using *PolySketch++* is better than using *PolySketch*. It can reduce 96.7%, 91.6% and 89.3% tile workload compared with using *PolySketch*. This makes the algorithm more efficient. For example, if we use *PolySketch* in huge polygons, we will get a huge number of tiles but most of them will be discarded because they did not overlap with others. By using *PolySketch++*, we will get fewer tiles at the beginning and discard most of them by checking very fewer tiles. For the remaining tiles, we will divide them into smaller tiles and check them again to reduce the LSI function. Therefore, we can use the hardware parallelism on the tiles which has a higher potential to have intersection point(s) with others and use less hardware resource to discard useless tiles. In addition, we can reduce more LSI function workload even though *PolySketch* has already used a very small tile size (such as 14 line segments). If the data sets are very large and we use a very small tile size (such as 10 line segments), the GPU may be out of memory. By using *PolySketch++*, we can divide the large tile into smaller tiles whose size is 4 line segments and do not have memory problems.

Table 6.5: The run-time of PolySketch (P) and PolySketch++ (P++) using large polygons (around 25000 vertices)

	P 32	P 64	P 128	P 256	P++ 128-4	P++ 256-4	P++ 256-8
Run-time(ms)	51	109	228	443	45	51	60

Table 6.5 shows that the performance of using random static size is not good. By using *PolySketch*, we need to run some experiments to choose a good tile size for the polygons. By using *PolySketch++*, we do not need to try different tile sizes and it can handle the situation by using child tile size. For the experiments using different tile sizes, we used the maximum threads possible based on the workload.

6.4.5 Execution Time Comparison Using Different GPUs

Table 6.6: The run-time of PolySketch and PolySketch++ with LSI function using Titan V and Titan Xp

	Ocean Static 256	Ocean Static 25	Ocean Adaptive 512-4	Ocean Adaptive 256-4
Titan V(s)	3.54	0.60	0.38	0.39
Titan Xp(s)	6.86	0.78	0.43	0.46

Table 6.6 is about the run-time of *PolySketch* and *PolySketch*++ for LSI function using Titan V and Titan Xp. For the *PolySketch*++ columns, the first number is the parent tile size and the second number (after the hyphen) is the child tile size. Titan V is superior to Titan Xp which results in a difference in performance.

6.5 Rectangle Intersection Filter on GPU

6.5.1 Overview of LMBR Filter

After reading the input data, the first step in our HiFiRe system is to create candidate tasks. Each task includes two polygons whose MBRs overlap with each other. Generally, we build R-tree for all MBRs of a layer and search MBRs of another layer to create candidate tasks, which are sequential on CPU. In this section, we will introduce how we parallelize this step and create candidate tasks on GPU.

In the HiFiRe system, parallelizing the step of creating candidate tasks on GPU should be efficient as well considering time performance, especially for large data size. The naive method is the all-to-all method, which means each MBR of one layer should be compared with all MBRs of another layer. However, there are some

methods to improve it, such as space partition. The paper [73] introduces the PRI-GC algorithm to solve rectangle intersection problem by using GPU plus CPU.

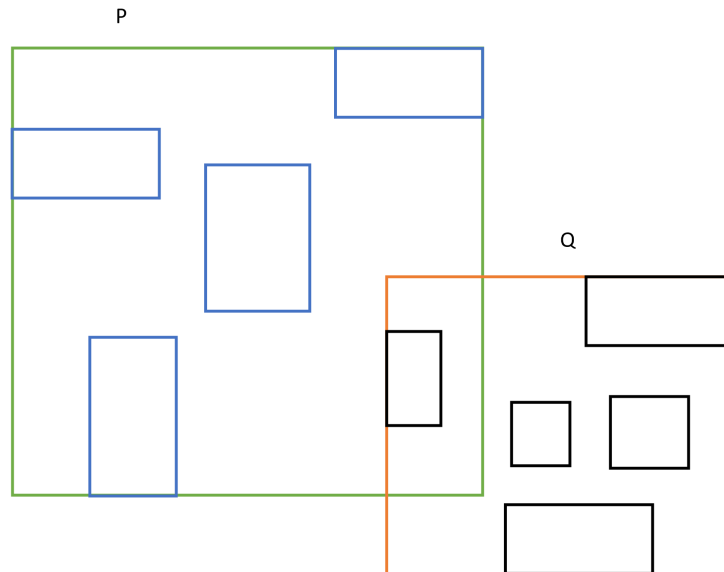


Figure 6.7: An example of LMBR

There are some challenges of space partition: (1) If one MBR overlap with more than one cell, this MBR should be stored in more than one cell, which is duplication and may cause duplicate results. It may cause extra work to be calculated. (2) Which data structure should be used to store different numbers of MBRs overlapping with different cells? Managing GPU memory is different from CPU. This challenge will affect GPU performance. (3) Some cells would not include any MBR, which should be still checked. Therefore, we develop a new parallel method of rectangle intersection on GPU based on the data partition. We call it larger-MBR (LMBR).

The basic idea of LMBR is to partition data according to contiguous MBRs considering the location stored in the array instead of space partition. We will get some new MBRs and each new MBR will be calculated based on and contain some contiguous MBRs. LMBR size means that how many MBRs the larger MBR

includes. However, in the real dataset, the location of a polygon stored in the array can be very far from its contiguous polygon. This will cause the larger MBR to contain a lot of dead space where does not contain any MBR. To solve this problem, we will sort MBRs to make sure every contiguous MBR in the array is also close to its adjacent MBR considering their locations.

Figures 6.7 shows an example of LMBR. Blue and black rectangles are adjacent MBRs of different layers after sorting. P is the new larger MBR that contains 4 real MBRs and Q contains 5 real MBRs. Since P and Q intersect, we will check all 4 MBRs of P with all 5 MBRs of Q. If a new larger MBR does not overlap with other larger MBRs, we can ignore all MBRs within this LMBR.

6.5.2 The Implementation of LMBR Filter

There are three steps of the LMBR method: (1) sort MBRs, (2) get new larger MBRs of contiguous MBRs, and (3) check which larger MBRs overlap with others and use real MBRs to get correct results. Each MBR has two points which are the lower left point and upper right point to represent it. We sort all MBRs based on the x coordinate of the lower left point of each MBR. Other values related to MBRs should be reordered according to the sort. We use CUDA thrust library [74] to sort the values. Figures 6.8 shows the pseudo-code of LMBR filter applied to rectangle intersection function using CUDA.

One advantage of LMBR is there would not be any duplication because every larger MBR is based on MBRs which it includes. Therefore, we do not need to deal with how to handle duplicate results and extra calculation work on duplicate polygons. This makes the algorithm on GPU more efficient. Another advantage is that as much as calculation work can be parallelized, which takes advantage of the powerful calculation capability of GPU.

```
1  //create different blocks for different LMBRs of layer 1
2  __global__ void LMBR_bothLayers(int numLmbrL2,
3      int lmbrSize1,int lmbrSize2,int *prefixSum1,
4      int *prefixSum2,MBR *lmbr1,MBR *lmbr2,
5      MBR *realMBR1,MBR *realMBR2,...){
6    int i = blockIdx.x;
7    //numLmbrL2 is the number of LMBRs of layer 2
8    for (int j=threadIdx.x; j<numLmbrL2; j+=blockDim.x){
9        if(lmbr1[i] overlaps lmbr2[k]){
10         for(int ii=0; ii<lmbrSize1;ii++){
11           for(int jj=0; jj<lmbrSize2;jj++){
12             if(realMBR1[prefixSum1[i]+ii] overlaps
13                             realMBR2[prefixSum2[j]+jj]){
14               //Store rectangle intersection pairs
15               RI(prefixSum1[i]+ii, prefixSum2[j]+jj);
16             }
17           }
18         }
19       }
20     }
21   }
22 }
23
24 int createTasks(MBR *realMBR1,MBR *realMBR2,...){
25     ...
26     //use CUDA Thrust to sort MBRs of both layers
27     //copy data from CPU to GPU
28     //calculate larger MBRs of both layers
29     //and related values
30     LMBR_bothLayers <<<numLmbrL1, numThreads>>> (
31     numLmbrL2, lmbrSize1, lmbrSize2, prefixSum1,
32     prefixSum2, lmbr1, lmbr2, realMBR1, realMBR2,...);
33     ...
34 }
35
```

Figure 6.8: CUDA Implementation of LMBR Filter

6.6 LMBR Filter Performance

Table 6.7: The execution time of using R-tree, all-to-all method or LMBR filter

	R-tree	All-to-all	LMBR
Water(s)	2.27	0.315	0.029
Lakes(s)	40.38	130.93	0.281
Parks(s)	72.36	177.82	0.494

Table 6.7 shows the execution time of using R-tree, all-to-all method or LMBR function. The Parks dataset means that Parks (includes around 9.7M polygons) and Sports (includes around 1.8M polygons). We can see the LMBR method works better than the general R-tree method or all-to-all method. For the Water, Lakes and Parks datasets, we can reach 78x, 143x and 146x speedup compared with using R-tree.

Table 6.8: Time performance of LMBR by partitioning both layers and using different LMBR sizes

LMBR size	1024	512	256	128	64	32
Water(ms)	472	191	102	63	43	33
Lakes(ms)	2598	1308	736	449	321	281
Parks(ms)	3831	2125	1295	843	593	494

Table 6.9: Time performance of LMBR by partitioning 1 layer and using different LMBR sizes

LMBR size	1024	512	256	128	64	32
Water(ms)	91	59	43	33	29	37
Lakes(ms)	1085	806	712	782	1376	2815
Parks(ms)	2020	1532	1467	1531	2643	5492

Table 6.8 and Table 6.9 show the performance of LMBR filter applied to rectangle intersection by using different LMBR sizes and partitioning both layers or

only 1 layer. The execution time of LMBR function also includes the time of sorting MBRs and storing candidate tasks. Table 6.8 is about partitioning both layers. We can see that using a smaller LMBR size works better than a larger LMBR size. Table 6.9 is about partitioning the first layer. We can see that 256 or 128 works better than other LMBR sizes. In addition, considering the best time performance, we can get more benefit from partitioning both layers for larger datasets. For the Water dataset, the best performance of partitioning 1 layer is similar to the best performance of partitioning both layers.

6.7 Conclusion

We have developed an adaptive filter for geometric intersection computations by using CUDA dynamic parallelism. We have shown that a dynamic tiling based filter performs better than a static tiling based filter for large and huge polygons. In static tiling, it is difficult to choose an optimal tile-size to reduce workload and the best performance depends on the manual tuning of the tile-size. This is not a problem in *PolySketch*++ as we have demonstrated in the experiments. Using a two-step filter, dynamic tiling adjusts the tile-size adaptively to minimize the workload. To get the best performance, our Hierarchical Filter and Refine (HiFiRe) system can now use a hybrid approach with static and dynamic tiling. We believe other spatial computing workloads will benefit from leveraging GPU dynamic parallelism by using our work as an exemplar. For rectangle intersection computation, we also develop the LMBR method on GPU.

Chapter 7
Conclusion and Future Work

7.1 Conclusion

In this dissertation, we have developed a hierarchical filter and refinement system for parallel geometric intersection operation, which can be used to spatial join or polygon overlay. Efficient filters have been developed to reduce actual computation in refinement, such as PolySketch, PolySketch++, PSCMBR, PNP related filters, and so on.

PolySketch is a representation of a spatial object by a set of tiles and each tile is a subset of consecutive vertices of a geometry. We extend the classical filter and refine strategy by introducing PolySketch technique. In general, there are not many segment intersections in polygon overlay [7], so even when the CMBR is less effective, PolySketch can provide significant workload reduction. In addition, the strengths of CMBR and PolySketch can be combined to get better performance so we have developed the PSCMBR filter. PolySketch++ is an adaptive filter that improves PolySketch by making the tile-size adaptive. PNP filter and refine algorithms are also important in the HiFiRe system. We extend the filter and refine technique by adding and using different efficient filters for speeding up LSI and PNP operations.

7.2 Future Work

Although the capacity of a single GPU can be very powerful, it is also worth exploring to improve and implement algorithms by using multi-GPUs over large-scale datasets. According to the paper [75], a single GPU works well when the graphs fit into GPU's memory but it requires multi-GPUs while achieving higher performance and/or scaling to larger graphs. In addition, we can integrate our

GPU-accelerated system to MPI-GIS and MapReduce implementations which we have built as an HPC system for geospatial analytics [76, 7, 77, 19, 78].

Since the size of spatial data is increasing at a huge scale and many real applications are still generating a huge amount of spatial data, analyzing and querying spatial data require high-performance computing (HPC) infrastructures [79]. The performance is mainly constrained in the computational capacity in spatial analysis systems or infrastructures. Therefore, improving such systems and infrastructures and developing HPC algorithms are helpful to big data management and analysis that can be used to solve different real research problems. For example, processing medical images, analyzing remote sensing data, spatial econometrics analyses, GIS, and so on [4, 80].

BIBLIOGRAPHY

[1] E. T. Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freiref, "GPU rasterization for real-time spatial aggregation over arbitrary polygons," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, pp. 352–365, 2017.

[2] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?" *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1661–1673, 2018.

[3] J. P. Singh, Y. K. Dwivedi, N. P. Rana, A. Kumar, and K. K. Kapoor, "Event classification and location prediction from tweets during disasters," *Annals of Operations Research*, pp. 1–21, 2017.

[4] S. K. Prasad, D. Aghajarian, M. McDermott, D. Shah, M. Mokbel, S. Puri, S. J. Rey, S. Shekhar, Y. Xe, R. R. Vatsavai *et al.*, "Parallel processing over spatial-temporal datasets from geo, bio, climate and social science communities: A research roadmap," in *2017 IEEE International Congress on Big Data (BigData Congress).* IEEE, 2017, pp. 232–250.

[5] J. Zhang, S. You, and L. Gruenwald, "Large-scale spatial data processing on GPUs and GPU-accelerated clusters," *Sigspatial Special*, vol. 6, no. 3, pp. 27–34, 2015.

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

[7] S. Puri and S. K. Prasad, "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, 2015, pp. 576–585.

[8] G. Greiner and K. Hormann, "Efficient clipping of arbitrary polygons," *ACM Transactions on Graphics (TOG)*, vol. 17, no. 2, pp. 71–83, 1998.

[9] E. L. Foster, K. Hormann, and R. T. Popa, "Clipping simple polygons with degenerate intersections," *Computers & Graphics: X*, p. 100007, 2019.

[10] K. G. Pillai, R. A. Angryk, and B. Aydin, "A filter-and-refine approach to mine spatiotemporal co-occurrences," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2013, pp. 104–113.

[11] J. H. University, https://gisanddata.maps.arcgis.com/apps/opsdashboard/index.html#/bda7594740fd40299423467b48e9ecf6.

[12] DailyMail, https://www.dailymail.co.uk/news/article-8164301/Government-tracking-Americans-cell-phones-spread-coronavirus.html.

[13] B. R. Vati, "A generic solution to polygon clipping," *Communications of the ACM*, vol. 35, no. 7, pp. 56–64, 1992.

[14] A. Paudel and S. Puri, "OpenACC Based GPU Parallelization of Plane Sweep Algorithm for Geometric Intersection," in *International Workshop on Accelerator Programming Using Directives.* Springer, 2018, pp. 114–135.

[15] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 1, p. 7, 2007.

[16] H. Veenhof, P. Apers, and M. Houtsma, "Optimisation of spatial joins using filters," in *Advances in Databases, 13th British National Conference on Databases, Manchester, United Kingdom.* Springer, 1995, pp. 136–154.

[17] D. M. Mount, "Geometric intersection," in *Handbook of Discrete and Computational Geometry, chapter 33.* Citeseer, 1997.

[18] M. I. Shamos and D. Hoey, "Geometric intersection problems," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976).* IEEE, 1976, pp. 208–215.

[19] S. Puri, D. Agarwal, X. He, and S. K. Prasad, "MapReduce algorithms for GIS polygonal overlay processing," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum.* IEEE, 2013, pp. 1009–1016.

[20] S. You, J. Zhang, and L. Gruenwald, "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*, 2016, pp. 42–49.

[21] J. Zhang and S. You, "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, pp. 23–32.

[22] S. Puri and S. K. Prasad, "Output-sensitive parallel algorithm for polygon clipping," in *2014 43rd International Conference on Parallel Processing.* IEEE, 2014, pp. 241–250.

[23] J. Nievergelt and F. P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Communications of the ACM*, vol. 25, no. 10, pp. 739–747, 1982.

[24] S. V. Magalhães, M. V. Andrade, W. R. Franklin, and W. Li, "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*, 2015, pp. 45–54.

[25] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz, "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, no. 11. NIH Public Access, 2012, p. 1543.

[26] P. van Oosterom, "An R-tree based map-overlay algorithm," in *Proc. EGIS*, vol. 94, 1994, pp. 318–327.

[27] S. K. Prasad, M. McDermott, X. He, and S. Puri, "GPU-based Parallel R-tree Construction and Querying," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 618–627.

[28] J. A. Orenstein, "Redundancy in spatial databases," in *ACM SIGMOD Record*, vol. 18, no. 2. ACM, 1989, pp. 295–305.

[29] M. McKenney and T. McGuire, "A parallel plane sweep algorithm for multi-core systems." in *GIS*, 2009, pp. 392–395.

[30] A. Margalit and G. D. Knott, "An algorithm for computing the union, intersection or difference of two polygons," *Computers & Graphics*, vol. 13, no. 2, pp. 167–183, 1989.

[31] A. Aji, G. Teodoro, and F. Wang, "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*. ACM, 2014, pp. 15–20.

[32] S. K. Prasad, S. Shekhar, M. McDermott, X. Zhou, M. Evans, and S. Puri, "GPGPU-accelerated interesting interval discovery and other computations on geospatial datasets: A summary of results," in *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. ACM, 2013, pp. 65–72.

[33] B. Donnelly and M. Gowanlock, "A coordinate-oblivious index for high-dimensional distance similarity searches on the gpu," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

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

[35] T. Brinkhoff, H.-P. Kriegel, and R. Schneider, "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, 1993, pp. 40–49.

[36] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger, "Multi-step processing of spatial joins," *Acm Sigmod Record*, vol. 23, no. 2, pp. 197–208, 1994.

[37] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: the international journal for geographic information and geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.

[38] D. Aghajarian, S. Puri, and S. Prasad, "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, 2016, p. 18.

[39] D. Aghajarian and S. K. Prasad, "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, 2017, p. 56.

[40] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

[41] N. Beckmann and B. Seeger, "A revised r*-tree in comparison with related index structures," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 799–812.

[42] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[43] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 237–246, 1993.

[44] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," *ACM Sigmod Record*, vol. 25, no. 2, pp. 259–270, 1996.

[45] D. Sidlauskas, S. Chester, E. T. Zacharatou, and A. Ailamaki, "Improving spatial data processing by clipping minimum bounding boxes," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 425–436.

[46] Y. Gao, B. Wu, J. Luo, and H. Qiu, "GPU-based arbitrary polygon intersection area algorithm," *DEStech Transactions on Engineering and Technology Research*, no. ismii, 2017.

[47] N. Ferreira, M. Lage, H. Doraiswamy, H. Vo, L. Wilson, H. Werner, M. Park, and C. Silva, "Urbane: A 3d framework to support data driven decision making in urban development," in *2015 IEEE conference on visual analytics science and technology (VAST)*. IEEE, 2015, pp. 97–104.

[48] J. R. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational geometry*, vol. 22, no. 1-3, pp. 21–74, 2002.

[49] G. Zimbrao and J. M. De Souza, "A raster approximation for processing of spatial joins," in *VLDB*, 1998, pp. 558–569.

[50] B. Simion, S. Ray, and A. D. Brown, "Surveying the landscape: An in-depth analysis of spatial database workloads," in *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, 2012, pp. 376–385.

[51] F. Wang, A. Aji, and H. Vo, "High performance spatial queries for spatial big data: from medical imaging to GIS," *Sigspatial Special*, vol. 6, no. 3, pp. 11–18, 2015.

[52] C. Gao, F. Baig, H. Vo, Y. Zhu, and F. Wang, "Accelerating Cross-Matching Operation of Geospatial Datasets using a CPU-GPU Hybrid Platform," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 3402–3411.

[53] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: optimizing GPU memory bandwidth via warp specialization," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–11.

[54] D. Li, H. Wu, and M. Becchi, "Exploiting dynamic parallelism to efficiently support irregular nested loops on GPUS," in *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, 2015, pp. 1–1.

[55] H. Wu, D. Li, and M. Becchi, "Compiler-assisted workload consolidation for efficient dynamic parallelism on GPU," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 534–543.

[56] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *2015 IEEE 31st international conference on Data Engineering*. IEEE, 2015, pp. 1352–1363.

[57] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce," in *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 6, no. 11. NIH Public Access, 2013.

[58] S. Shekhar, S. Ravada, V. Kumar, D. Chubb, and G. Turner, "Parallelizing a gis on a shared address space architecture," *Computer*, vol. 29, no. 12, pp. 42–48, 1996.

[59] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang, "Sparkgis: Resource aware efficient in-memory spatial query processing," in *Proceedings of the 25th ACM SIGSPATIAL international conference on advances in geographic information systems*, 2017, pp. 1–10.

[60] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*, 2015, pp. 1–4.

[61] A. Eldawy, M. F. Mokbel *et al.*, "The era of big spatial data: A survey," *Foundations and Trends® in Databases*, vol. 6, no. 3-4, pp. 163–273, 2016.

[62] H. Vo, A. Aji, and F. Wang, "Sato: a spatial data partitioning framework for scalable query processing," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2014, pp. 545–548.

[63] "CGAL - Intersecting Sequences of dD Iso-oriented Boxes," https://doc.cgal.org/latest/Box_intersection_d/index.html.

[64] A. Zomorodian and H. Edelsbrunner, "Fast software for box intersections," in *Proceedings of the sixteenth annual symposium on Computational geometry*, 2000, pp. 129–138.

[65] W. R. Franklin, C. Narayanaswaml, M. Kankanhalll, D. Sun, M.-C. Zhou, and P. Y. Wu, "Uniform grids: A technique for intersection detection on serial and parallel machines," in *Proceedings of Auto-Carto 9*. Citeseer, 1989.

[66] P. Schneider and D. H. Eberly, *Geometric tools for computer graphics.* Elsevier, 2002.

[67] S. Ray, C. Higgins, V. Anupindi, and S. Gautam, "Enabling numa-aware main memory spatial join processing: An experimental study," *ADMS@ VLDB*, 2020.

[68] Y. Liu, J. Yang, and S. Puri, "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, 2019, pp. 141–151.

[69] J. Yang and S. Puri, "Efficient parallel and adaptive partitioning for load-balancing in spatial join," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2020, pp. 810–820.

[70] D. Li, H. Wu, and M. Becchi, "Nested parallelism on GPU: Exploring parallelization templates for irregular loops and recursive computations," in *2015 44th International Conference on Parallel Processing.* IEEE, 2015, pp. 979–988.

[71] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled kernel launch for dynamic parallelism in GPUs," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2017, pp. 649–660.

[72] J. Zhang, A. M. Aji, M. L. Chu, H. Wang, and W.-c. Feng, "Taming irregular applications via advanced dynamic parallelism on GPUs," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018, pp. 146–154.

[73] S.-H. Lo, C.-R. Lee, Y.-C. Chung, and I.-H. Chung, "A parallel rectangle intersection algorithm on gpu+ cpu," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.* IEEE, 2011, pp. 43–52.

[74] "CUDA Thrust," https://docs.nvidia.com/cuda/thrust/index.html.

[75] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU graph analytics," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2017, pp. 479–490.

[76] D. Agarwal, S. Puri, X. He, and S. K. Prasad, "A system for GIS polygonal overlay computation on linux cluster-an experience and performance report," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.* IEEE, 2012, pp. 1433–1439.

[77] S. Puri, A. Paudel, and S. K. Prasad, "MPI-Vector-IO: Parallel I/O and partitioning for geospatial vector data," in *Proceedings of the 47th International Conference on Parallel Processing, ICPP*, 2018, p. 13.

[78] S. Puri, D. Agarwal, and S. K. Prasad, "Polygonal overlay computation on Cloud, Hadoop, and MPI," *Encyclopedia of GIS*, pp. 1–9, 2015.

[79] S. K. Prasad, M. McDermott, S. Puri, D. Shah, D. Aghajarian, S. Shekhar, and X. Zhou, "A vision for GPU-accelerated parallel computation on geo-spatial datasets," *SIGSPATIAL Special*, vol. 6, no. 3, pp. 19–26, 2015.

[80] X. He, Y. Tao, Q. Wang, and H. Lin, "Multivariate spatial data visualization: a survey," *Journal of Visualization*, vol. 22, no. 5, pp. 897–912, 2019.