Efficient PRAM and Practical GPU Algorithms for Large Polygon Clipping with Degenerate Cases

Buddhi Ashan M. K.
Department of Computer Science
University of Texas at San Antonio
buddhiashan.mallikakankanamalage@utsa.edu

Satish Puri
Department of Computer Science
Marquette University
satish.puri@marquette.edu

Sushil K. Prasad
Department of Computer Science
University of Texas at San Antonio
sushil.prasad@utsa.edu

2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 979-8-3503-0119-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/CCGRID57682.2023.00060

***Abstract*—Polygonal geometric operations are fundamental in domains such as Computer Graphics, Computer-Aided Design, and Geographic Information Systems. Handling degenerate cases in such operations is important when real-world spatial data are used. The popular Greiner-Hormann (GH) clipping algorithm does not handle such cases properly without perturbing vertices leading to inaccuracies and ambiguities. In this work, we parallelize the $\mathcal{O}(n^2)$-time general polygon clipping algorithm by Foster et al., which can handle degenerate cases without perturbation. Our CREW PRAM algorithm can perform clipping in $\mathcal{O}(\log n)$ time using $n + k$ number of processors with *simple* polygons, where n is the number of input edges and k is the number of edge intersections. For efficient GPU implementation, we employ three effective filters which have not been used in prior work on polygon clipping: 1) Common-minimum-bounding-rectangle filter, 2) Count-based filter, and 3) Line-segment-minimum-bounding-rectangle filter. They drastically reduce $\mathcal{O}(n^2)$ candidate edge pairs comparisons by 80%-99%, leading to significantly faster parallel execution. In our experiments, C++ CUDA-based implementation yields up to 40X speedup over real-world datasets, processing two polygons with a total of 174K vertices on an Nvidia Quadro RTX 5000 GPU compared to the sequential Foster's algorithm running on an Intel Xeon Silver 4210R CPU.**

***Index Terms*—polygon clipping, degenerate intersections, Greiner-Hormann algorithm, Foster et al. algorithm, GPU algorithm, PRAM algorithm**

I. Introduction

Polygons are used to represent boundaries of regions or objects in Geographic Information Systems (GIS) and Computer Graphics domains. Geometric set operations such as intersection, union, and set difference on very large polygonal datasets are common and important in both of these domains. A polygon is a 2-dimensional closed geometric region constructed with three or more straight line segments which are connected at their starting and ending points. Line segments are referred to as edges. The starting and ending points of those line segments are referred to as vertices. There are different types of polygons: 1) *simple* polygon where its edges do not self-intersect, 2) $self\text{-}intersection$ polygon where some edges self-intersect, 3) *convex* polygon where all interior angles are no more than $180°$, and 4) *concave* polygon where some interior angles are greater than $180°$. General polygon clipping algorithms can handle all these types of polygons.

In general, polygon clipping refers to the calculation of $P \cap Q$ between two polygons P and Q. But clipping algorithms can be modified to compute other geometric set operations such as union and set difference [1]. Polygon intersection involves calculating an output polygon which is a common region between the input polygons (Fig. 1). The output intersection polygon can also be a collection of polygons depending on the inputs (Fig. 2). A typical approach for this calculation involves checking the intersection between each edge pair, as in the Greiner-Hormann (GH) algorithm. This task is compute-intensive and can take several minutes for a single pair of polygons. Real-world datasets involve two layers of polygons, which runs over hours. We limit the scope of this paper to handling a pair of polygons.

Degenerate Intersections: A degenerate intersection in polygon clipping consists of having at least one vertex which lies on the edge of the other polygon or sharing a vertex between input polygons. These intersections can be found with any type of polygon. Consider Fig. 1a with polygon P and polygon Q. Vertex P_2 of P lies on edge (Q_1, Q_2) of Q and vertex P_4 of P lies on edge (Q_3, Q_4) of Q. Fig. 1b and 1c show two possible approximate clipping results for P and Q based on the perturbing direction. But the correct intersecting area as shown in Fig. 1a is bounded by $[I_1, P_2, I_2, I_3, P_4, P_5, I_4]$ vertices. The additional region included or missing in the intersection region with the perturbing technique vastly affects the accuracy of GIS polygonal applications. Fig. 2 has additional examples of degenerate cases including overlaps.

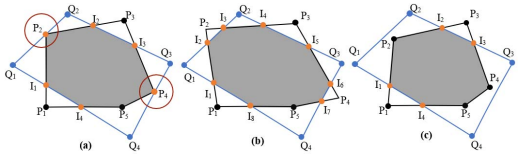

Fig. 1. (a): Vertex P_2 and P_4 lie on edges of Q, hence they are degenerate vertices. (b), (c): Two possible clipped output regions produced by GH algorithm depending on the perturbing direction. Gray color denotes clipped output

Vatti's algorithm and GH algorithm are two well-known algorithms for general polygon clipping [2], [3]. They start the clipping process by discovering the intersection vertices, inserting them into the input polygons, labeling them, and finally traversing them to generate the output polygon(s) using the labels generated previously. Given two polygons with n

979-8-3503-0119-9/23/$31.00 ©2023 IEEE
DOI 10.1109/CCGrid57682.2023.00060

number of vertices in total, the GH algorithm has a time complexity of $\mathcal{O}(n^2)$. Vatti's algorithm is output-sensitive since the execution time depends on the number of output edge intersections (k). Vatti's algorithm uses $\mathcal{O}(n \log n)$ time sweep-line algorithm and the GH algorithm uses brute force $\mathcal{O}(n^2)$-time intersection finding by testing all edge pairs from the input polygons [4], [5]. However, in the case of self-intersecting polygons, Vatti's algorithm needs self-intersection points to be reported from each input polygon apriori. This overhead is not present in the GH algorithm, which can build output polygons without computing the self-intersections explicitly. For this reason, the GH algorithm can outperform Vatti's algorithm for self-intersecting polygons [2]. The GH algorithm is unable to properly handle degenerate cases without using perturbing methods, which can lead to incorrect and ambiguous results as shown in Fig. 1b and 1c. Degenerate cases are common in real-world polygonal datasets and especially GIS applications can suffer from inaccurate results due to perturbing methods [1].

Foster et al. [1] proposed an extension to the GH algorithm, introducing more sophisticated labeling to handle degenerate cases properly. It runs in $\mathcal{O}(n^2)$ time, where time is dominated by the intersection calculation phase similar to the GH algorithm.

In this work, we chose Foster's algorithm for developing a GPU clipping implementation because it is more amenable to GPU parallelization when compared to plane-sweep-based Vatti's algorithm [4], [5]. The current state-of-the-art parallel GH algorithm runs in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n + k)$ processors, where k is the number of intersections, on the CREW (Concurrent Read Exclusive Write) PRAM (Parallel Random Access Memory) model for $simple$ polygons without properly handling degenerate cases [5].

To optimize working space to store the intersections which in practice are a small fraction of the worst-case $\mathcal{O}(n^2)$, we break down our GPU intersection point calculation step into first finding the counts of intersections for each edge, then allocating space, storing the intersections, and finally sort the appended intersections in each edge. We introduce three filters in our practical implementation, eliminating the bulk of line segment intersection computations.

In practice, there is only a small percentage of intersecting edges for a given pair of polygons. A key challenge is to quickly find those before invoking a sufficiently-involved edge-to-edge intersection algorithm which has to deal with multiple scenarios including degenerate cases. The Minimum Bounding Rectangle (MBR) of a two-dimensional figure is the smallest rectangle encapsulating it, with the sides of the rectangle parallel to either the x or y axis (Fig. 8 and Fig. 9). The Common Minimum Bounding Rectangle (CMBR) is the intersection of two MBRs (Fig. 8). Our CMBR Filter (CMF) is a linear time filter that can reduce the quadratic workload in the intersection calculation phase by a good fraction by eliminating those edges which do not intersect with the CMBR of input polygons (Fig. 8). Such edges can provably not contribute to any intersections between the two polygons. Aghajarian et al. employed the CMBR idea for spatial join problem [6]. Puri et al.'s GPU polygon clipping algorithm [5] does not leverage any filters.

Our Line Segment MBR Filter (LSMF) is a simple spatial test to determine whether the MBRs of the two given line segments overlap or not (Fig. 9c). The MBR overlap test is computationally much faster than the elaborated line segment intersection algorithm. Therefore, this filter can eliminate a large number of non-intersecting edge pairs efficiently. With intersection counts available, our Count-based Filter (CF) simply eliminates those edges which are determined to have no intersections in the storing step. In our experiments, these filters collectively improved speedup by up to 9.5 times.

The main contributions of this work are as follows:

- A CREW PRAM polygon clipping algorithm that handles all degenerate cases and runs in $\mathcal{O}(\log n)$ time for polygons without self-intersections using $\mathcal{O}(n+k)$ number of processors, where n is the total number of edges in the input polygons, and k is the number of edge intersections.
- A CUDA C++ implementation of parallel Foster's algorithm which can handle degenerate cases.
- Our GPU algorithm outperforms the GPU polygon clipping presented in [5] with the help of reduced workload using MBR based filters. These filters drastically eliminate $\mathcal{O}(n^2)$ candidate edge pairs by 80%-99%, leading to significantly faster parallel execution.
- Our GPU algorithm yields up to 40 times speedup on real-world datasets by processing two polygons with 174K vertices in total on an Nvidia Quadro RTX 5000 GPU, compared to the sequential Foster's clipping algorithm [1] executing on an Intel Xeon Silver 4210R CPU.

The rest of the paper is organized as follows. Section II discusses the background of polygon clipping algorithms and their limitations. Section III presents PRAM Foster's polygon clipping algorithm and its time complexity analysis. Section IV presents our GPU polygon clipping algorithm. Section V presents an experimental analysis of the GPU algorithm. Section VI presents our conclusions and future work.

II. Background

A. Polygon Clipping

There are well-known polygon clipping sequential and parallel algorithms that are used in Computer Graphics and GIS domains. Maillot's algorithm only clips using a rectangle, but not against polygons [7]. Sutherland-Hodgman, Weiler-Atherton, Liang-Barsky, Vatti's, and Greiner-Hormann algorithms can clip a concave polygon against another concave polygon [2], [3], [8]–[10]. However, Vatti's and Greiner-Hormann algorithms stand out since they can clip arbitrary polygons [2], [3].

Apart from Vatti's algorithm, plane-sweep based sequential polygon clipping algorithms are also discussed in [11]–[13]. The GH algorithm has a simpler way to represent polygons than Vatti's and its time complexity is not output sensitive.

There are parallel clipping algorithms based on [8], [10] implemented on classic parallel architectures [14]. GPU accelerated plane-sweep based clipping algorithms are harder to

implement since they use complex tree-based data structures such as parallel segment tree and hierarchical plane-sweep tree (array of trees) [15]. Naïve $\mathcal{O}(n^2)$ algorithms and grid partitioning have been used in practical GPU overlay algorithms discussed in [15], [16]. There is a multicore Vatti's algorithm implementation presented in [4]. [17] discusses a plane-sweep based practical GPU polygon clipping algorithm without using self-balancing tree structures. Parallel many-core and multicore implementations of the GH algorithm are presented in [5]. Both GPU clipping algorithms discussed in [5], [15] are unable to handle degenerate cases.

Parallel polygon clipping algorithm research lacks an algorithm that can handle complex situations such as degenerate cases taking advantage of faster GPUs. In the current literature, the existing systems are either single node sequential algorithms [1], heterogeneous algorithms which can not handle degenerate cases [5], [15] or compute cluster-based (CPU only) using MapReduce/MPI that do not harness GPUs [4], [18].

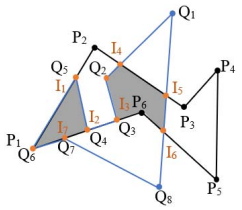

Fig. 2. Polygons P and Q intersection. P_1 Q_3, Q_4, Q_5, Q_6, *and* Q_7 are degenerate vertices. The resulting polygon consists of 2 polygons; $[P_1, I_1, I_2]$ and $[I_4, I_5, I_6, P_6, I_3, Q_2]$. I_4, I_5, and I_6 are X-intersections.

B. Limitations of the GH Algorithm

GH polygon clipping algorithm is simple and works with all types of polygons. But, the GH algorithm can produce incorrect results when handling degenerate cases. This is due to the even-odd rule violation in the intersection labeling phase of the GH algorithm when handling degenerate cases [2]. The GH algorithm can handle degenerate cases by perturbing such intersections. But this leads to inaccurate and ambiguous results depending on the direction of the perturbation. Foster et al. [1] address this issue by introducing additional labels.

C. Foster's Polygon Clipping Algorithm

Similar to the GH algorithm, Foster's algorithm uses two doubly-linked lists to represent the input polygons. Each vertex of a polygon has links to previous, next, and neighbor vertices. $Neighbor$ refers to the same intersection vertex saved in the two input polygons (Fig. 2 and Fig. 3). Intersection vertices and source vertices can be identified using the $intersection$ and $source$ vertex properties.

The algorithm has three major steps, 1) intersection point calculation, 2) intersection vertex labeling, and 3) result tracing. In Foster's algorithm, the intersection point calculation phase remains the same as in the GH algorithm. It employs all-to-all edge intersection computation to find intersecting edge pairs and the intersection vertices. Foster's algorithm adds more advanced labels to the intersection vertices apart from $entry/exit$ labels used in the GH algorithm (Fig. 3 depicts these vertices using EN/EX labels for the polygons in Fig. 2). New labels help to handle degenerate cases properly.

1) Intersection point calculation: This phase aims to identify the intersecting edge pairs and save the intersection vertices in the correct location of the two input polygons. Degenerate intersections and overlaps reuse the source vertices and mark them as intersections (e.g., I_3 and Q_3, I_2 and Q_4, and P_1 and Q_6 in Fig. 2 and Fig. 3). The Intersection vertices in the input polygons are linked as neighbors (for example, I_1 and Q_5 in Fig. 2). Intersection classification is further discussed in our PRAM algorithm in the next section.

2) Intersection vertex labeling: The GH algorithm only uses the $entry/exit$ label (based on the inside/outside status of polygonal edges with respect to another polygon), where an entry intersection is always followed by an exit vertex and vice versa. But, this hypothesis is only valid with non-degenerate cases. The degenerate cases need more information to produce correct results. The intersection vertex labeling phase in Foster's algorithm consists of multiple stages involving different classifications to identify the intersecting label. The three stages of labeling are described in the next section along with our PRAM algorithm. These labels help to correctly classify the $entry/exit$ label.

3) Trace result: The result tracing starts from a crossing intersection and traverses the polygons. It starts from one polygon and can switch between considering the $exit/entry$ label and the geometric set operation.

Fig. 2 shows an example polygon P and Q clipping using the intersection operation. The list of degenerate intersections in this example is P_1, Q_3, Q_4, Q_5, Q_6, and Q_7. Foster's algorithm prepares the data structure in Fig. 3 at the end of the intersection point calculation phase. The data structure consists of two doubly-linked lists, where the source vertices of polygon P and Q are saved with non-degenerate intersections (denoted with I in Fig. 3). The arrows denote the next and previous links. Dotted lines denote $neighbor$ links. Assume the result trace starts from I_1 of P. The tracing order is $Q_4\ of\ Q, I_7\ of\ P$, and P_1. I_7 is removed in the final result since (I_2, I_7) and (I_7, Q_6) edges overlap with (P_6, P_1) edge. For the next clipping region, assume trace starting from I_4 of P. The tracing order is $I_5\ of\ Q, I_6, P_6\ of\ P, I_3$, and $Q_2\ of\ Q$. The traversal change between P and Q is based on the geometric set operation (intersection in this case). $Neighbor$ links are used to switch between polygon vertex lists.

III. PRAM Foster's Clipping Algorithm

A. Terminology

We use the following terminology to explain polygon clipping and our parallel algorithms. The input consists of two polygons P and Q, where set of edges of P are $E^P = \{e_1^P, e_2^P, \cdots, e_i^P, \cdots, e_n^P\}$ and set of edges of Q are $E^Q = \{e_1^Q, e_2^Q, \cdots, e_i^Q, \cdots, e_m^Q\}$. Set of vertices of P are

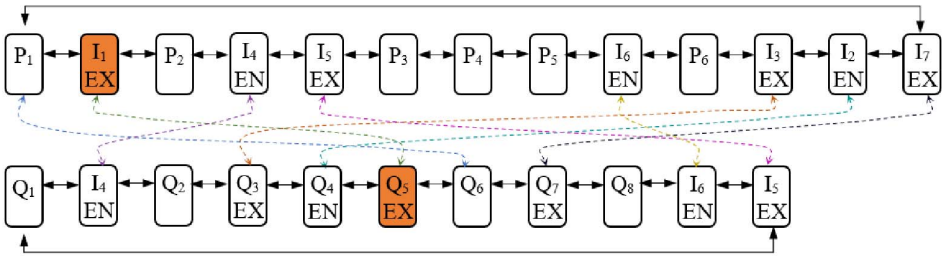

Fig. 3. Foster's clipping algorithm uses doubly-linked lists to represent polygons including intersection vertices. Non-degenerate intersections are added as new vertices in the relevant edge in the sorted order by α. Degenerate vertices are not duplicated. But they are linked with a new vertex or a degenerate vertex depending on the case. This intersection graph is a visual representation of the vertices of polygons in Fig. 2. The arrows denote the next and previous property between vertices. Dotted lines denote $neighbor$ links between the same intersection vertex, but located in different polygons. The $entry/exit$ labels are shown for $crossing$ and $delayed\ crossing$ (orange color node) intersections. The resulting output consists of 2 polygons; $[I_1(Q5), I_2(Q4), P_1(Q6)]$ and $[I_4, I_5, I_6, P_6, I_3, Q_2]$

$V^P = \{v_1^P, v_2^P, \cdots, v_i^P, \cdots, v_n^P\}$ and set of vertices of Q are $V^Q = \{v_1^Q, v_2^Q, \cdots, v_i^Q, \cdots, v_m^Q\}$. $n \in \mathbb{N}$, $n \geq 3$, $v_i^P \in \mathbb{R}^2$, and $e_i^P = (v_i^P, v_{i+1}^P)$, $i \in \{1, \cdots, n-1\}$, $e_n^P = (v_n^P, v_1^P)$. The parent vertex of an edge e_i^P is the first vertex of that edge, v_i^P. Similarly, m, v_i^Q, e_i^Q, i, and e_n^Q are defined in terms of polygon Q.

- Intersection vertex: $v_i^C = (x_i^C, y_i^C)$
- Alpha value (α_i^P): Relative position of v_i^C in e_i^P [1]
- Beta value (β_i^Q): Relative position of v_i^C in e_i^Q [1]
- Number of intersections: k

B. PRAM Polygon Clipping Algorithm

Algorithm 1 sketches our PRAM algorithm with three major steps: intersection point calculation, intersection vertex labeling, and result tracing. Sequential intersection point calculation is a simple but costly operation. Intersection point calculation only needs the vertex data of the contributing two edges which are locally available to each thread. Since the calculated intersections are appended in parallel, edge-wise sorting is necessary to maintain the order of intersections as they appear on each edge.

Intersections are of three types: X-intersection, T-intersection (Fig. 4a and Fig. 4b), and V-intersection (Fig. 4c). Similarly, overlapping edge intersections are of three types: X-overlap (Fig. 5a), T-overlap (Fig. 5b, Fig. 5c), and V-overlap (Fig. 5d). These labels depend on α and β values [1]. The α and β values (between 0 and 1) are used to denote the relative position of an intersection point within the two endpoints of the two intersecting line segments that generated the intersection point. Since α and β values are computed for each pair of line segment intersections, these aforementioned intersections and overlap type information can be independently computed using local information and therefore can be parallelized for all intersections. This is computed in the first step of polygon clipping *Algorithm 1* when line segment intersections are computed. Local sorting of intersection points within an edge using α and β values is carried out for all edges. The data structure for polygonal vertices is implemented using an array representation of a linked list. The test for α and β values for labeling are the same as in Foster's algorithm. In Step 1, prefix sum is used to insert new intersection vertices and pack the intersections with their α/β values consecutively, assuming a 2D matrix for storage of intersection information.

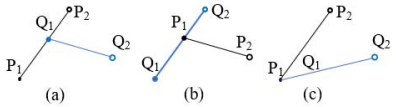

Fig. 4. Degenerate intersection types [1]. T-intersections (a, b) and V-intersection (c).

Intersection vertex labeling consists of three major stages, as shown in *Algorithm 1*. In the $first\ stage$, the non-overlapping intersections are labeled $Crossing$ (Fig. 1: I_1, I_2, I_3, and I_4) or $Bouncing$ (Fig. 1: P_2 and P_4) depending on the relative location of the contributed edge from the polygon Q. Each intersection vertex is linked with the before and after edge from both polygons. Overlapping intersections are labeled $Left|On$, $Right|On$, $On|On$, $On|Left$, or $On|Right$ depending on the relative turn of contributed edge from the polygon P (see examples in Fig. 6). We calculate the signed areas of triangles whose vertices are chosen from three consecutive vertices $p1$, $p2$, $p3$, and a test point q and use these areas to determine a left or right turn. Therefore, $crossing$ and $bouncing$ labels can be independently in parallel because the area calculation based on neighborhood information is locally available.

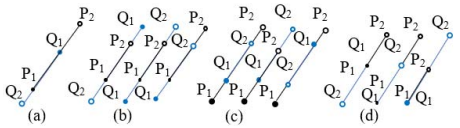

Fig. 5. Degenerate overlap types [1]. X-overlap (a), T-overlap (b, c), and V-overlap (d).

In the $second\ stage$ (Step 2.b), the intersection chains with adjacent overlapping edges are labeled. An $intersection\ chain$ starts with a turn label in $x|On$ format and ends with $On|y$, where $x, y \in \{right, left\}$. Intersections within the chain are labeled as $On|On$. If neighbors of an

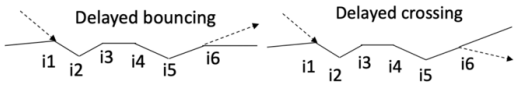

Fig. 6. Overlapping edges ($i1$, $i2$, .. , $i6$) from P and Q are shown as an $intersection\ chain$. i) Delayed bouncing intersection chain: leftmost dotted line is labeled $Left|On$. The rightmost dotted line is labeled $On|Left$. ii) Delayed crossing intersection chain: leftmost dotted line is labeled $Left|On$. The rightmost dotted line is labeled $On|Right$. The vertices $i1$ to $i6$ are labeled $On|On$.

intersection vertex are also intersections and labeled with a turn label of $On|On$, they are considered within a chain. An intersection with $x|On$ label is marked as a starting vertex of an intersection chain if the next vertex is labeled $On|On$. An intersection is at the end of an intersection chain if its label is $On|y$ and the label of the previous vertex is $On|On$.

Handling delayed intersection chains: In order to determine the inside/outside status of an overlapping polygonal edge with respect to another polygon, intersection chains are classified as $delayed\ bouncing$ and $delayed\ crossing$. An example of an intersection chain is shown in Fig. 6. We save starting ($i1$) and ending ($i6$) vertices in two arrays in the order they appear in the polygons. The corresponding locations in the arrays provide the start and end of a particular intersection chain, which are labeled $delayed\ crossing$ or $delayed\ bouncing$ depending on x and y names. Intersection chain labeling computation can be reduced to segmented prefix sum [19]. Segmented prefix sum is a variation of prefix sum where the input is an array of labeled intersection vertices with an auxiliary array of boolean values to denote segment boundaries on which prefix sum should be performed. Intersection vertices with $x|On$ and $On|y$ are marked with True value and those with $On|On$ labels are marked with False value to denote segment boundaries. The operator for the prefix sum is '+' which is used to calculate the start and end index of an intersection chain.

In the $third\ stage$ (Step 2.c), the $crossing$ intersections are labeled. The $entry/exit$ label is computed using the point-in-polygon test. Following the rules of Foster's algorithm, endpoints of a $delayed\ bouncing$ are marked with $entry/exit$ labels in the same way as regular $crossing$ intersection vertices. The endpoints of a $delayed\ crossing$ are marked either both as $entry$ or both as $exit$ points [1]. The remaining steps of $entry/exit$ labeling in parallel are similar to the method described in [5]. Finally in Step 3, the application of the Link Nullification rule from [5] removes the non-contributing parts of the input polygons. The remaining parts in the intersection graph between $entry$ vertex and $exit$ vertex are contributing. The contributing parts form the output polygon(s).

C. Time Complexity Analysis

We assume $n \geq m$. Logarithmic time complexity is achievable using n^2 number of processors on a CREW PRAM model. The number of processors used by the line segment intersection reporting algorithm can be made output-sensitive by invoking PRAM algorithms from [20], [21]. An output-sensitive algorithm can determine the number of intersections (output size k) online and performs better when the output size (k) is relatively small. Using an output-sensitive algorithm for reporting line segment intersections leads to $\mathcal{O}(\log n)$ time using $\mathcal{O}(n + k/\log n)$ number of processors where k is the number of intersections for $simple$ polygons [5], [20] and in $\mathcal{O}((n+k)\log n \log\log(n)/p)$ time where p is the number of processors and $p \leq (n+k)$ [5], [21] when self-intersecting polygons are used.

Algorithm 1 - PRAM Foster's Polygon Clipping

Input: $P(V^P)$, $Q(V^Q)$. V^P and V^Q are vertex arrays. E^P and E^Q are edge arrays. $|E^P|{=}n$, $|E^Q|{=}m$, and $n \geq m$

1: Intersection point calculation
   a: Calculate pairwise edge intersections from Polygon P and Q and store them in array V^C, $v_l^C{=}e_i^P \cap e_j^Q$ where i and j denote edge index for P and Q polygons.
   b: For each intersection point, using coordinates from e_i^P and e_j^Q, calculate α^P and β^Q values. Using α^P and β^Q values, classify the intersection type and overlap type.
   c: Insert (v_l^C, α^P) in V^P following v_i^P and (v_l^C, β^Q) in V^Q following v_j^Q.
   d: Sort the array of non-degenerate intersections in each edge e_i^P and e_j^Q based on α^P and β^Q values.
2: Intersection vertex labeling
   a: Each v_i^C is given a label based on $first\ stage$ labeling rules.
   b: $Second\ stage$ labeling to classify intersection chains.
   c: $Third\ stage$ labeling based on $entry/exit$ labels for $crossing$ intersections.
3: Tracing labeled intersection graph for output construction.

Sorting in Step 1 can be done in $\mathcal{O}(\log k)$ time using Cole's merge sort algorithm [5], [22]. In the worst case scenario $k{=}\mathcal{O}(n^2)$ and it takes $\mathcal{O}(\log n)$ time. Steps 1.c and 1.d are implemented using logarithmic time parallel prefix sum and parallel sorting algorithms.

Step 2.b consists of adjacent overlapping edge intersection chain labeling. Using a parallel segmented prefix sum algorithm, this step takes $\mathcal{O}(\log n)$ time and $\mathcal{O}(n+k)$ processors.

Step 3 can be performed in logarithmic time complexity using $O(n+k)$ processors by utilizing PRAM list ranking and PRAM point-in-polygon test as shown in Lemma 1 to 4 from [5].

The time complexity of our PRAM algorithm is dominated by Step 1.a. Its time complexity is $\mathcal{O}(\log n)$ time using $\mathcal{O}(n + k/\log n)$ number of processors for $simple$ polygons. This time complexity is the same as in the CREW PRAM GH algorithm in [5], which does not handle degenerate cases properly, whereas *Algorithm 1* does.

IV. CUDA-based Practical Implementation

The most expensive computation of Foster's algorithm is the pairwise edge intersection operation. In our experiments, it consumes 99% of the execution time (Fig. 7). Therefore,

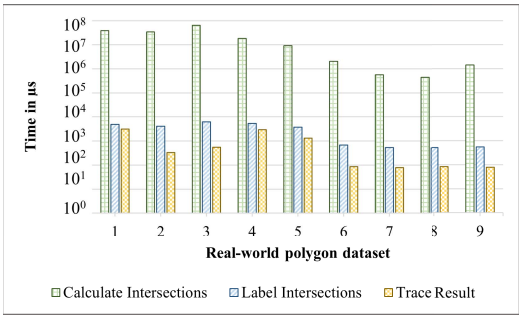

Fig. 7. Sequential time breakdown for major steps in Foster's algorithm using different datasets. The datasets are described in Table I.

our parallel algorithm is focused on this step to improve the overall efficiency and scalability of the clipping algorithm.

A. GPU Data Structures

All GPU data structures employed are arrays. Each input polygon is copied into GPU global memory using two arrays, one for x coordinates and the other for y coordinates. Intersection counts for edges are calculated locally by each thread and saved in $Count$ arrays. Steps 1.b and 1.c use these $count$ arrays to prepare exclusive prefix sum arrays. Their sizes are equal to input polygon sizes + 1. The $neighborMap$ array contains a mapping of contributing polygon P ID at the index of contributing polygon Q ID. The IS^P and IS^Q arrays are used to save the vertices of polygons including any discovered non-degenerate intersection vertices. The NB^P and NB^Q arrays link $neighbors$. For a given intersection $v_l^C{=}e_i^P \cap e_j^Q$, the $nb_i^P{=}j$ and $nb_j^Q{=}i$. The NB arrays are used to read neighbors in $\mathcal{O}(1)$ time.

Algorithm 2 - Count Intersections

Input: $P(V^P)$, $Q(V^Q)$, $CMBR = MBR_p \cap MBR_Q$.
Output: $Count_1$, $Count_2$.

1: **for each** $GPUBlock_i, 0 \leq i \leq |E^P|$ **do in parallel**
2: **if** $CMBR\ intersects\ e_i^P$ **then**
3: **for each** e_j^Q **do**
4: **if** $MBR_{e_i^P}\ intersects\ MBR_{e_j^Q}$ **then**
5: **if** $e_i^P\ intersects\ e_j^Q$ **then**
6: $Count_1[i]$ += 1
7: **if** $e_i^P \cap e_j^Q$ not $degenerate\ case$ **then**
8: $Count_2[i]$ += 1
9: **end if**
10: **end if**
11: **end if**
12: **end for**
13: **end if**
14: **end for**

B. GPU Polygon Clipping Algorithm

Our GPU algorithm is sketched as *Algorithm 5*. Intersection calculation only requires vertex data of the contributing edges which are locally available for each thread. But there are a few challenges due to the algorithm parallelization and the limitations of the GPU hardware. The challenges are: appending the non-degenerate intersection vertices in the contributing edges in parallel, degenerate intersections reuse source vertices as intersections, link both degenerate and non-degenerate intersections across input polygons, parallel non-degenerate intersection vertex append does not guarantee the correct order of the intersections as they appear on the edges, dynamic memory allocation in the GPU memory is inefficient.

Algorithm 3 - Create MapQ List

Input: $P(V^P)$, $Q(V^Q)$, PS_1^Q, PS_2^Q. **Output:** $neighborMap$.

1: **for each** $GPUBlock_i, 0 \leq i \leq |E^Q|$ **do in parallel**
2: **if** $PS_1^Q[i] \neq PS_1^Q[i+1]$ **then**
3: **for each** e_j^P **do**
4: **if** $MBR_{e_i^Q}\ intersects\ MBR_{e_j^P}$ **then**
5: **if** $e_i^Q\ intersects\ e_j^P$ **then**
6: **if** $e_i^P \cap e_j^Q$ not $degenerate\ case$ **then**
7: $lcount$ += 1
8: **else**
9: $lcount = 0$
10: **end if**
11: $neighborMap[PS_2^Q[i] + lcount] = j$
12: **end if**
13: **end if**
14: **end for**
15: **end if**
16: **end for**

To mitigate these challenges, new data structures ($count$, $prefix\ sum$, and $neighborMap$ arrays) are introduced in step 1 along with four kernels: i) $CountIntersections$ followed by Thrust library based prefix sum calculation [23], ii) $Create\ MapQ\ List$, iii) $Save\ Intersection\ Vertices$, and iv) $Sort\ Q$.

Step 1.a uses $Count\ Intersection$ kernel (*Algorithm 2*) where each thread computes $e_i^P \cap E^Q$. This configuration helps to easily balance the load while maintaining a higher GPU utilization since the input polygons are larger. The $Count_1$ and $Count_2$ arrays provide the total intersection counts which help to allocate GPU memory for the new intersection vertices in the input polygons. *Algorithm 2* also takes advantage of the faster GPU-shared memory. When the polygon Q is sufficiently smaller to fit in the shared memory, each thread takes advantage of the faster memory in the $e_i^P \cap E^Q$ computation by first copying the V^Q collaboratively from global memory to the shared memory. When Polygon Q is too large to fit in the shared memory, it is tiled and threads iteratively copy a tile of the polygon Q collaboratively into the shared memory and complete the intersection calculation for each tile ($e_i^P \cap \hat{E}^Q,\ \hat{E}^Q \subset E^Q$). This is carried out until all threads finish working on polygon Q completely.

We compute two exclusive prefix sums for each input polygon in Step 1.b: i) $PS_1^{P,Q}$: prefix sum of intersection counts

Algorithm 4 - Save Intersection Vertices

Input: $P(V^P)$, $Q(V^Q)$, $neighborMap$, PS_1^P, PS_2^P, PS_1^Q, PS_2^Q. **Output:** IS^P, IS^Q, NB^P, NB^Q.

1: **for each** $GPUBlock_i, 0 \leq i \leq |E^P|$ **do in parallel**
2: **if** $PS_1^Q[i] \neq PS_1^Q[i+1]$ **then**
3: **for each** e_j^Q **do**
4: **if** $MBR_{e_i^P}$ *intersects* $MBR_{e_j^Q}$ **then**
5: **if** e_i^P *intersects* e_j^Q **then**
6: $count \mathrel{+}= 1$
7: $(v_i^C, \alpha, \beta) \leftarrow e_i^P \cap e_j^Q$
8: **if** $e_i^P \cap e_j^Q$ not *degenerate case* **then**
9: $count2 \mathrel{+}= 1$
10: Add (v_i^C, α) *to* IS^P
11: **else**
12: $count2 = 0$
13: Add $(e_i^P, 0)$ *to* IS^P
14: **end if**
15: **if** $e_i^Q \cap e_j^P$ not *degenerate case* **then**
16: Add (v_i^C, β) *to* IS^Q
17: **else**
18: Add $(e_j^Q, 0)$ *to* IS^Q
19: **end if**
20: $start = PS_2^Q[j]$
21: $end = PS_2^Q[j+1]$
22: **for** $k = start$ *to* $k = end$ **do**
23: **if** $i == neighborMap[k]$ **then**
24: $nIndex = k + 1$
25: $NB^P[PS_2^P[i]+count2] = nIndex$
26: $NB^Q[k] = PS_2^P[i]+count2 + 1$
27: **end if**
28: **end for**
29: **end if**
30: **end if**
31: **end for**
32: **end if**
33: **end for**

of each edge, and ii) $PS_2^{P,Q}$: prefix sum of non-degenerate intersections including parent vertex. Each thread uses $PS_1^{P,Q}$ and $PS_2^{P,Q}$ to save non-degenerate intersections in the correct location of the input polygon. Degenerate intersections involve a source vertex and a new vertex to be inserted in an input polygon unless it is a V-intersection / overlap. The new vertex is inserted, treating it as a non-degenerate intersection vertex and the related source vertex is marked as an intersection.

In step 1.d $Create\ MapQ\ List$ kernel (*Algorithm 3*) is used to map the indices of an intersection vertex in polygon P with Q. The map helps to find the exact location of an intersection vertex in the polygon Q array by saving the contributing edge ID of the polygon P in the $NeighborMap$ array. This kernel can be replaced with a reduction function to make the neighbor connection between polygon P and Q. But it is more expensive with heavy communication and synchronization involved. Another alternative is to duplicate

Algorithm 5 - GPU Foster's Polygon Clipping

Input: $P(V^P)$, $Q(V^Q)$
where $|E^P|$=n, $|E^Q|$=m, and $n \geq m$

1: Intersection point calculation
a: For each e_i^P: save the count of all intersections ($Count_1$) and non-degenerate intersections ($Count_2$).
b: Compute exclusive prefix sum of $Count_1$ (PS_1^P) and $Count_2$ (PS_2^P) for polygon P.
c: Compute PS_1^Q and PS_2^Q similarly for polygon Q.
d: Save indices of intersection vertices from polygon Q in $neighborMap$ array
e: Save intersections in IS^P and IS^Q arrays with neighbor links in NB^P and NB^Q.
f: Sort non-degenerate intersections within a source edge e_i^P of IS^P based on α values. Similarly, sort IS^Q based on β values.
2: Intersection vertex labeling
a: Classify V^C using rules from $first\ stage$ labeling.
b: Classify V^C using rules from $second\ stage$ labeling in sequential manner on CPU side.
c: Classify V^C using rules from $third\ stage$ labeling in sequential manner on CPU side.
3: Trace labeled intersection graph to construct output polygons in a sequential manner on the CPU side.

intersection computation using $e_i^Q \cap E^P$. But it is inefficient for larger polygons.

Step 1.e uses $Save\ Intersection\ Vertices$ kernel (*Algorithm 4*) to save the intersection vertices in the IS^P and IS^Q arrays at the correct indices which are calculated using the prefix sum and $NeighborMap$ arrays. The kernel searches the current edge ID of the polygon P in $NeighborMap$ to calculate $nIndex$. $nIndex - 1$ is the exact location to save a non-degenerate intersection. The search starts from the starting vertex of the current edge from polygon Q, which is available from PS^Q. *Algorithm 4* also saves neighbor links in the NB^P and NB^Q arrays.

Step 1.f needs a barrier between IS^P and IS^Q sorting since it affects the $neighbor$ links. Therefore, we sort IS^P at the end of $Save\ Intersection\ vertices$ kernel and then employ $Sort\ Q$ kernel to sort IS^Q. This provides a global synchronization between the two sortings. Intersections are sorted edge-wise using α/β values as the key. We employ parallel Radix sort in each thread to sort non-degenerate intersection vertices within a particular source edge.

The Intersection vertex labeling consists of three stages as explained in the PRAM algorithm. We only perform $first\ stage$ labeling on the GPU side using $Initial\ label$ kernel (Step 2.a). Classification is based on v_i^C's previous, next, and $neighbor$ vertices along with v_i^C's neighbor's next and previous vertices. These are locally available to each thread in IS^P, IS^Q, NB^P, and NB^Q arrays.

Our experiments show that real-world data consume less than 1% of the total execution time in labeling and result

tracing phases together (Fig. 7). Therefore, we have not yet parallelized these final phases and we leverage Foster's algorithm sequentially on the CPU side.

C. GPU filtering of line segment intersection tests

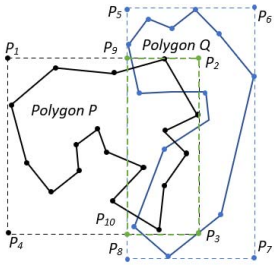

Fig. 8. MBR^P=[P_1, P_2, P_3, P_4], MBR^Q=[P_5, P_6, P_7, P_8]. $CMBR$=[P_9, P_2, P_3, P_{10}]. All edges of P and Q are intersected with the CMBR and only those intersecting are used for further processing.

We introduce three filters, 1) CMBR Filter (CMF), 2) Count-based Filter (CF), and 3) Line Segment MBR Filter (LSMF). They help to improve the efficiency of the duplicated all-to-all-edge intersection computations in step 1 by filtering out non-intersecting edge pairs using relatively inexpensive tests. Step 1.a. uses CMF and then LSMF. Step 1.d. and 1.e. use CF followed by LSMF.

1) CMF: CMF can improve the efficiency of the parallel intersection point count phase. Aghajarian et al. used CMF to solve the spatial join problem [6]. CMF first calculates the two Minimum Bounding Rectangles (MBRs) for the two input polygons and calculates a Common MBR (CMBR) for them. CMBR is the rectangle that is common to both MBRs (see Fig. 8). Next, the algorithm filters out any edge that does not intersect with the CMBR (*Algorithm 2*). Initially, we have n number of input edges. CMF reduces n to $\hat{n}$ where $\hat{n} \leq n$. Now, the required work for the algorithm has been reduced to $\hat{n}^2$. This filter has $\mathcal{O}(n)$ amount of work, which leads to faster parallel execution using up to n threads.

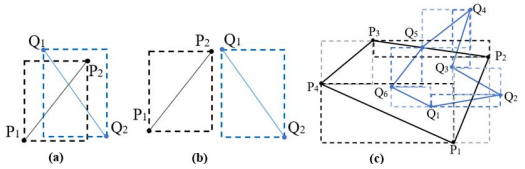

Fig. 9. Illustration of LSMF: (a) possible intersection, (b) no intersection possible, (c) MBRs drawn for edges of the input polygons. Overlapping MBRs nominated for intersection candidacy.

2) LSMF: Calculating and classifying an intersection vertex is a relatively expensive operation involving many condition checking. We optimize this task by employing LSMF, which evaluates if a given pair of edges is spatially close enough to possibly intersect with each other (see Fig. 9). The filter treats the edge coordinates as their MBRs and checks if there is an overlap between the MBR pair (*Algorithm 2, Algorithm 3, Algorithm 4*). As shown in Fig. 9c, overlapping MBR pairs are reported as intersection candidates and the rest are discarded. Candidate edge pairs are then subjected to a refined calculation to check if there is an intersection. For example, in Fig. 9c, $e_4^P(p_4^P, p_1^P)$ has a MBR overlap with $e_6^Q(p_6^Q, p_1^Q)$, but the edges do not intersect with each other. Since real-world data have only a few intersecting edge pairs, most of the pairs get eliminated. LSMF provides a cheaper way to eliminate the vast majority of edge pairs (see Table III).

3) CF: In the first stage, we check if $e_i^P \cap E^Q$ discovers any intersections by checking the intersection count for this edge using $PS_{i+1}^P - PS_i^P$. Count value 0 implies there are no intersections to be discovered for e_i^P and further processing is unnecessary (*Algorithm 3, Algorithm 4*). If the count is a positive integer, there are that many intersections required to be discovered and saved in the intersection arrays. In the second stage of CF, we use PS^Q to filter any edge candidates of Q.

V. Experimental Results

1) Testbed: We used an Intel Xeon Silver 4210R CPU-equipped workstation running on 2.40GHz with 64 GB of memory and an Nvidia Quadro RTX 5000 GPU card with 16 GB of VRAM, 48 SMs, and 3072 CUDA cores. We used CUDA 11 and C++ for our GPU implementation.

2) Dataset: The experiments exhibit two major aspects of this work: 1) the ability to handle degenerate intersections, and 2) weak scalability. We used two datasets to evaluate the performance of the GPU clipping algorithm compared to Foster's clipping algorithm. The first dataset only consists of real-world polygons. The second dataset is a synthetic dataset which consists of manipulated real-world polygons and simulated polygons. The number of intersections found in real-world polygons is smaller than the total input vertices. The synthetic dataset has test cases to evaluate a larger number of intersections.

We used Classic [24], Ocean (ne_10m_ocean), Land (ne_10m_land), and Continents [25] real-world datasets for our experiments. The nine test cases consist of large polygons as shown in Table I along with their corresponding polygon ID, the number of vertices, degenerate and non-degenerate intersection sizes, parallel, and sequential run times, and speedups. The result is the output polygon using the intersection operation.

The Synthetic test cases 1 to 3 in Table II represent k=$\mathcal{O}(n)$. We generated them using the Open-Street-Map lakes dataset (from http://spatialhadoop.cs.umn.edu/datasets.html) and Classic dataset polygons (S and C) [24]. These cases use a real-world polygon as the first polygon and a generated polygon as the second polygon, with the same size, maintaining k=$\mathcal{O}(n)$. Synthetic test case 4 to 8 in Table II represent k=$\mathcal{O}(n^2)$. In each of them, we generated two polygons of the same size based on the worst-case polygon shape in [2].

3) Comparison with the sequential Foster's algorithm: [1]

[1]We used the C++ implementation of Foster's polygon clipping algorithm, downloaded from the second author's webpage at https://www.inf.usi.ch/hormann/polyclip/.

TABLE I
PERFORMANCE OF OUR GPU POLYGON CLIPPING ALGORITHM ON REAL-WORLD DATASETS (EXECUTION TIMES EXCLUDING I/O TIMES) [24], [25].

#	Dataset	|P|	|Q|	|Result|	Degenerate count	Non-degenerate count	Sequential Time (ms)	Parallel Time (ms)	Speedup
1	Classic S, C	101,242	72,997	50,312	4	43	38,429	955	40
2	ne 10m ocean (36), ne 10m land (1)	66,475	66,475	66,447	66,474	0	34,602	1,041	33
3	ne_10m_ocean (0), ne_10m_land (4)	100,612	81,511	5	81,495	2	64,434	1,543	42
4	ne_10m_ocean (0), continents (521)	100,612	16,205	37,608	0	10,082	18,026	355	51
5	ne_10m_ocean (0), continents (1661)	100,612	12,613	16,895	2	1,425	9,048	252	36
6	ne_10m_ocean (2742), continents (1048)	15,547	15,653	43	0	18	2,050	136	15
7	ne_10m_ocean (2742), continents (1081)	15,547	4,562	20	0	14	561	134	4
8	ne_10m_ocean (2742), continents (1193)	15,547	4,028	33	0	32	439	132	3
9	ne_10m_ocean (2741), continents (1048)	10,887	15,653	156	0	32	1,447	136	11

TABLE II
PERFORMANCE OF OUR GPU POLYGON CLIPPING ALGORITHM ON A SYNTHETIC DATASET.

#	Polygon size	Number of Intersections	Sequential time (ms)	Parallel Time (ms)	Speedup
1	206,429	3.90×10^6	41,712,596	48,963	852
2	101,242	1.47×10^5	277,502	7,349	38
3	72,997	8.34×10^4	59,246	3,940	15
4	500	2.50×10^5	3,113	275	11
5	700	4.90×10^5	11,009	404	27
6	1,000	1.00×10^6	41,302	676	61
7	1,500	2.25×10^6	158,803	1,299	122
8	2,000	4.00×10^6	422,124	2,129	198

Using real-world dataset: We compared our CUDA C++ parallel implementation against Foster's sequential clipping algorithm employing intersection operation over nine test cases as shown in Table I. We first optimized the sequential Foster's implementation using LSMF for a fair comparison. The Other filters are more parallel friendly. We validated our results with Foster's algorithm for the correctness and calculated speedups for each test case, excluding I/O time. Table I shows the sequential times, parallel times, and speedups for each case. Since polygon clipping is output-sensitive, the speedups vary based on the number of intersections. The best speedup is shown in test case 1, where polygons with 170k and 70k are clipped. The resulting polygon has 50k vertices with only 47 intersections. Test cases 2 and 3 polygon sizes are smaller, but they have more intersections. Therefore, we observe a lower speedup than that of test case 1.

Using synthetic dataset: For k=$\mathcal{O}(n^2)$ (synthetic test cases 1 to 3), the best speedup is reported in synthetic test case 8. Simulated polygon sizes vary from 500 to 2000 each, which are smaller sizes than the polygon sizes used in synthetic test cases 1 to 3. However, they have all-to-all intersections, where the number of intersections ranges from 2.5×10^5 to 4×10^6. Synthetic test cases 1 to 3 also discover a similar number of intersections. Table II shows the sequential Foster's clipping algorithm run times against our GPU clipping algorithm. The best speedup for k=$\mathcal{O}(n)$ is reported in synthetic test case 1 reducing 11.5 hours of workload to 49 seconds, illustrating the effectiveness of our filters. In all cases, we can observe a positive correlation between input polygon size and the speedup, demonstrating the weak scalability of our parallel algorithm.

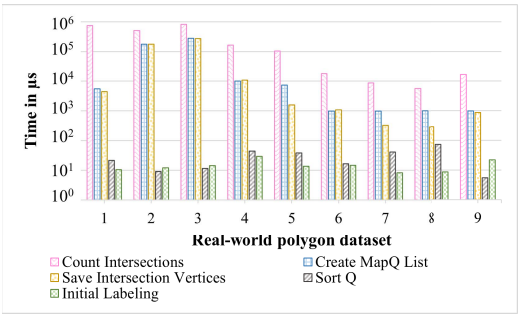

Fig. 10. Kernel execution timings for datasets in log scale.

4) Execution time breakdown in kernels: We can observe that for larger datasets the $Count\ intersections$ kernel is the dominating step in our parallel algorithm as discussed in the performance analysis. The highest total run times are observed in test cases 2 and 3. They have the highest number of input vertices among the test cases. The total run time in test case 1 is smaller than both of them, even though it has more input vertices. This is due to the low number of intersections. Fig. 10 is a detailed run-time breakdown for each kernel using the real-world dataset. Fig. 10 shows that $Create\ mapQ\ list$ and $Save\ intersection\ vertices$ kernels consume more time in test cases 2 and 3 compared to other cases. This is due to the higher number of intersections (see Table I). Both $Sort\ Q$ and $Initial\ labeling$ kernels only consume a smaller fraction of the execution time, including test cases 2 and 3. The majority of the intersections being

degenerate cases cause this observation since they do not need any sorting. *Initial labeling* maintains about 10 μs constantly for all test cases.

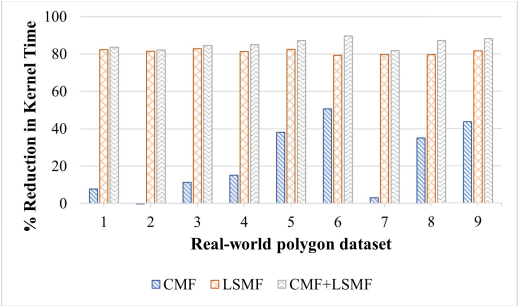

Fig. 11. % Reduction in *Count intersection* kernel run time after applying different filter configurations. Kernel time with no filters is considered 100%.

5) Filter performance: The *Count intersections* kernel employs CMF followed by LSMF. Both *MapQ* and *Save intersection vertices* kernels employ CF followed by LSMF. We compare both reduction in execution time (Fig. 11 and 12) and reduction in the number of candidate pairs eliminated (Table III) with and without filters in the aforementioned kernels. We use Equation 1 to calculate the percent reduction in run time considering filter-less kernel times as 100%.

$$saved\ time\ \% = 100 - \frac{time_{with\ filter}}{time_{no\ filter}} \times 100 \qquad (1)$$

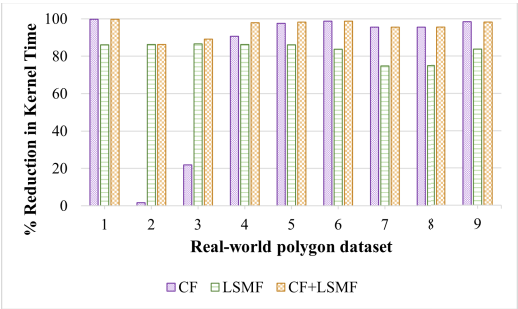

Fig. 12. % Reduction in *Map Q* kernel run time after applying different filter configurations. Kernel time with no filters is considered 100%.

Fig. 11 shows the percentage reduction in execution time using CMF and LSMF in the *Count intersections* kernel. In our test cases, CMF run-time savings vary in the $0\% - 50\%$ range. As seen in Table III, the low efficiencies are caused by the input polygons being located in a very close spatial proximity, where CMBR covers most of the edges of both polygons (Test case 1-4). It performs better in test cases 5-6 and 8-9. The efficiency of LSMF is $79\% - 82\%$ and in all the test cases, LSMF performs better than CMF. This is due to LSMF's more fine-grained filtering by only considering the spatial proximity of two edge pairs. CMF and LSMF together save $81\% - 88\%$ run time in the *Count intersections* kernel.

TABLE III
NO. OF SURVIVING CANDIDATE EDGE PAIRS AFTER EACH FILTER IN DIFFERENT KERNELS.

#	Total candidate edge pairs	Intersection count kernel		MapQ / Intersection save kernel	
		After CMF	After LSMF	After CF	After LSMF
1	7,390,362,274	6,073,522,515	25,599	1,936	54
2	4,418,925,625	4,418,925,625	206,710	123,825,380	206,706
3	8,200,984,732	6,746,910,003	252,427	6,641,679K	252,384
4	1,630,417,460	1,141,798,656	38,767	58,528,984	11,776
5	1,269,019,156	186,036,216	5,765	1,710,672	1,530
6	243,357,191	1,820,430	95	34	21
7	70,925,414	22,287,012	27	20	18
8	62,623,316	2,514,720	34	34	33
9	170,414,211	1,512,264	178	64	33

Fig. 12 shows the percentage reduction in execution time using CF and LSMF in the *Create mapQ List* kernel. Their efficiency range is $1\% - 99\%$. CF removes candidate edges with no intersections using the already calculated intersection counts by the *Count intersections* kernel. The large variance in the efficiency is due to the number of intersections discovered. Test cases 2 and 3 have the lowest reduction percentages since they report a large number of intersections (see Table III). All test cases report greater efficiency using CF than LSMF. Together, the savings range from $86\% - 99\%$.

The percentage reduction in execution time and the filter effectiveness using CF and LSMF in the *Save intersection vertices* kernel is similar to the *Create mapQ List* kernel.

VI. CONCLUSION

In this work, we present an efficient PRAM algorithm and a scalable GPU algorithm for polygon clipping which can also handle degenerate intersections. The state-of-the-art parallel GH algorithm is unable to handle degenerate intersections properly. We optimize our practical CUDA C++ based implementation by introducing three effective filters. CMF and LSMF have been used to solve spatial join problems in prior related work, but not for polygon clipping. The result is speeding up about 10 minutes of computation for large real-world polygons to about a second.

In the future, we plan on parallelizing the remaining steps of our parallel algorithm. Sequential post-processing only consumes a small run-time fraction for large polygons. But our experimental results suggest that there is still room for improvement. Effectively parallelizing the tracing results phase is challenging due to its sequential nature.

We also plan to extend this to large datasets by enabling clipping between two layers of polygons, a more real-world scenario. This needs an efficient method to identify candidate intersecting polygon pairs and balance the clipping workload across nodes. Here, our GPU clipping algorithm can be used as a subroutine for efficient pairwise polygon clipping.

REFERENCES

[1] E. L. Foster, K. Hormann, and R. T. Popa, "Clipping simple polygons with degenerate intersections," *Computers & Graphics: X*, vol. 2, p. 100007, 2019.

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

[3] B. R. Vatti, "A generic solution to polygon clipping," *Communications of the ACM*, vol. 35, no. 7, pp. 56–63, 1992.

[4] 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.

[5] 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.

[6] 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*, 2016, pp. 1–10.

[7] P. G. Maillot, "A new, fast method for 2d polygon clipping: analysis and software implementation," *ACM Transactions on Graphics (TOG)*, vol. 11, no. 3, pp. 276–290, 1992.

[8] I. E. Sutherland and G. W. Hodgman, "Reentrant polygon clipping," *Communications of the ACM*, vol. 17, no. 1, pp. 32–42, 1974.

[9] K. Weiler and P. Atherton, "Hidden surface removal using polygon area sorting," *ACM SIGGRAPH computer graphics*, vol. 11, no. 2, pp. 214–222, 1977.

[10] Y.-D. Liang and B. A. Barsky, "An analysis and algorithm for polygon clipping," *Communications of the ACM*, vol. 26, no. 11, pp. 868–877, 1983.

[11] L. J. Simonson, "Industrial strength polygon clipping: A novel algorithm with applications in vlsi cad," *Computer-Aided Design*, vol. 42, no. 12, pp. 1189–1196, 2010.

[12] 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.

[13] F. Martinez, A. J. Rueda, and F. R. Feito, "A new algorithm for computing boolean operations on polygons," *Computers & Geosciences*, vol. 35, no. 6, pp. 1177–1185, 2009.

[14] B.-O. Schneider and J. van Welzen, "Efficient polygon clipping for an simd graphics pipeline," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 3, pp. 272–285, 1998.

[15] 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*, 2013, pp. 304–313.

[16] 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.

[17] 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.

[18] 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.

[19] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[20] C. Rüb, "Computing intersections and arrangements for red-blue curve segments in parallel," *Proceedings of the Fourth Canadian Conference on Computational Geometry*, 1992.

[21] C. Rüb, "Line-segment intersection reporting in parallel," *Algorithmica*, vol. 8, no. 1, pp. 119–144, 1992.

[22] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

[23] 2022. [Online]. Available: https://docs.nvidia.com/cuda/thrust/index.html

[24] R. Modron, "Polygon clipping: a wrapper, a benchmark," 2011. [Online]. Available: https://rogue-modron.blogspot.com/2011/04/polygon-clipping-wrapper-benchmark.html

[25] 2022. [Online]. Available: https://www.naturalearthdata.com/

VII. ARTIFACT APPENDIX

A. Abstract

This artifact appendix provides the source code and dataset information to replicate the experiments demonstrated in this paper. The artifact contains bash scripts to conveniently execute the test cases for datasets including a template Excel file to generate tables and graphs shown in this paper.

B. Artifact check-list (meta information)

Program: C++ language

Compilation: C++ compiler

Dataset: The artifact requires a dataset of coordinates of polygons (at least a polygon pair). The dataset used for the experiments is in the data folder (poly_data.zip).

Run-time environment: Ubuntu 22.04

Execution: The GPU needs to be free while executing the experiments. Otherwise, it can affect the speedup of the parallel algorithm.

Metrics: The comparisons are based on the speedup metric. The formula for other metrics is mentioned in the paper.

Output: If there is any intersection between the input polygon pair, the output is a plain text file that contains the coordinates of the overlapping region of that polygon pair. It is saved in the results/ folder. In cases where the output contains a collection of polygons, their coordinates are split using ';'.

Experiments: Run the GPU algorithm, run Foster's sequential algorithm, and record execution times in the given spreadsheet to calculate speedup, and generate graphs.

Required disk space (approximately): 500MB

Time needed to prepare workflow (approximately): 20 minutes

Time needed to complete experiments (approximately): 15 hours

Publicly available:

DOI: https://doi.org/10.5281/zenodo.7683127

GitHub: https://github.com/buddhi1/GH-CUDA.git

C. Description

How to access: The source code is available in the main branch of the following GitHub repository. https://github.com/buddhi1/GH-CUDA.git

Hardware dependencies: This artifact requires a Linux x86 machine with an Nvidia GPU (We only tested on compute_70, compute_75, compute_80, and compute_86 architectures). To replicate the experiments with synthetic and real-world datasets, 500MB of free space is approximately required.

The specifications of the machine we used are as follows: Processor: Intel® Xeon(R) Silver 4210R CPU @ 2.40GHz, Memory: 64GB, GPU: NVIDIA Quadro RTX 5000 with 16GB memory.

Software dependencies: A Linux-based OS is required to build this artifact. Our GPU uses Nvidia driver version 520.

- GNU make

- C++ 11
- Cuda 11
- Thrust
- unzip

Dataset: There are multiple real-world and synthetic polygon data files. Each file consists of the (x, y) coordinates of a polygon. The first line represents the total number of vertices in the polygon. The data file path is data/poly_data.zip.

D. Installation

1) $ git clone https://github.com/buddhi1/GH-CUDA.git
2) $ cd GH-CUDA
3) Compile: $ make

make file reads the cuda path from $LD_LIBRARY_PATH variable. If the path is not correctly found, replace LIBCUDA variable in the make file with the correct path. Use the following commands to unzip data.

1) $ cd data/
2) $ unzip poly_data.zip
3) $ cd ..

Now data folder should have realworld and synthetic folders with data files in them.

Debug mode: Following steps can be used to quickly check if everything is set up correctly. case is the test_case id of real-world or synthetic dataset. Results are saved in the results/ folder.

1) Clean build: $ make clean
2) Compile: $ make
3) Real-world dataset experiments: (Fastest case=8)
   $ /bin/bash real-world_dataset_run.sh ⟨case⟩
4) Synthetic dataset experiments: (Fastest case=4)
   $ /bin/bash synthetic_dataset_run.sh ⟨case⟩
5) Clean sequential code:
   $ make cleanfoster
6) Compile sequential code:
   $ make foster
7) Real-world dataset experiments: (Fastest case=8)
   $/bin/bash real-world_dataset_foster.sh ⟨case⟩
8) Synthetic dataset experiments: (Fastest case=4)
   $ /bin/bash synthetic_dataset_foster.sh ⟨case⟩

E. Experimental workflow

How to run: Use save flag only when the resulting polygon needs to be saved to a file.

$ ./program ⟨base polygon file⟩ ⟨overlay polygon file⟩ ⟨result polygon path⟩ save

Demonstrated experiments have 3 cases. For convenience, we have included bash scripts to test each case. We recommend testing each case at a time.

1) Evaluate Performance: In this case, Foster's sequential algorithm and our parallel algorithm are executed on real-world and synthetic datasets.

a) Clean build: $ make clean
b) Compile: $ make
c) Real-world dataset experiments:
   $ /bin/bash real-world_dataset_run.sh
d) Synthetic dataset experiments:
   $ /bin/bash synthetic_dataset_run.sh
e) Clean sequential code:
   $ make cleanfoster
f) Compile sequential code:
   $ make foster
g) Real-world dataset experiments:
   $ /bin/bash real-world_dataset_foster.sh
h) Synthetic dataset experiments:
   $ /bin/bash synthetic_dataset_foster.sh

These executions generate real-world_tests.csv, synthetic_tests.csv, real-world_sequential.csv, and synthetic_sequential.csv spreadsheets in results/ folder with runtime breakdowns.

Copy real-world parallel execution data in real-world_tests.csv and real-world sequential data in real-world_sequential.csv to Table 1 in Real-world_dataset workbook of the template Excel file. Copy synthetic parallel execution data in synthetic_tests.csv and synthetic sequential data in synthetic_sequential.csv to the Table in Synthetic_dataset workbook of the template Excel file. The template Excel file is graphs.xlsx in the results/ folder.

Save results: Use save flag at the end of the script calls above to save clipped polygon coordinates in files. Foster's results are saved in optimizedFostersAlgorithm/results/ folder and GPU results are saved in results/ folder. Saved results can be used to compare outputs and we do not recommend using save flag for performance evaluation.

2) Evaluate Filter Performance: Three experiments need to be completed to compare different filter configurations.

CMF and CF filters only

a) Clean build: $ make clean
b) Compile: $ make cmfcf
c) Real-world dataset experiments:
   $ /bin/bash real-world_dataset_run.sh

These executions generate real-world_tests.csv, and synthetic_tests.csv spreadsheets in results/ folder. Save data in table 3 case ii of the template Excel.

LSMF filter only

a) Clean build: $ make clean
b) Compile: $ make lsmf
c) Real-world dataset experiments:
   $ /bin/bash real-world_dataset_run.sh

Save data in table 3 case iii of the template Excel.

No filters

a) Clean build: $ make clean
b) Compile: $ make nofilters
c) Rreal-world dataset experiments:
   $ /bin/bash real-world_dataset_run.sh

Save data in table 3 case iv of the template Excel. Case i uses the data from section A where all filters are employed.

Use that data in Table 3 case i. Table 4 and the reduction percentage graphs are auto-generated using Table 3 data.

3) Evaluate Edge Pair Reduction by Filters: We record the number of edge pairs after each filter which is employed in $Intersection\ Count$ and $Create\ Map\ Q\ List$ kernels.

a) Clean build: $ make clean
b) Compile: $ make count
c) real-world dataset experiments:
   $ /bin/bash real-world_dataset_run.sh

This execution generates real-world_tests_counts.csv spreadsheet in results/ folder with the edge pair counts after each filter for the above-mentioned kernels. Table 2 in the template excel represents this data.

F. Evaluation and expected results

This section describes how to replicate the results presented in the paper. We used an Excel sheet to visualize and tabulate our experimental data which is shared in the results/ folder (graphs.xlsx). We have supplied the necessary bash scripts that generate the execution times and other information in spreadsheets. To replicate the tables and the graphs in our paper, these tabulated data can be pasted to the corresponding tables.