# Existing Data Structure

point2D p;                  // coordinates of the vertex

  vertex\* prev;               // pointer to previous vertex

  vertex\* next;               // pointer to next vertex

  vertex\* neighbour;          // pointer to neighbouring vertex for intersection vertices

  bool source;                // to mark source vertices of the polygon

  bool intersection;          // to mark intersection vertices

  double alpha;               // to describe relative edge position of an intersection vertex

IntersectionLabel label;    // type of intersection vertex

  EntryExitLabel enex;        // entry/exit "flag"

# Data Structure for GPU

4 arrays to represent input data (Polygon P and Polygon Q) defined using pointers and malloc.

double [sizeP] polyPX  
double [sizeP] polyPY

double [SizeQ] polyQX  
double [sizeP] polyQY

2 arrays for intersection points (polygon P intersection point and polygon Q intersection point) and alpha value defined using malloc

intersectionsP[i] represents [x,y,alpha] distinct intersection points available at 3 times threadID

double [3\*sizeP\*sizeQ] intersectionsP  
double [3\*sizeP\*sizeQ] intersectionsQ

2 arrays to represent neighbors for intersectionsP and intersectionsQ

neighborP[i] represets [neighbor type]

neighbor types = {P1->0, Q1->1, I\_P->2, I\_Q->3}

int [sizeP\*sizeQ] neighborP  
int [sizeP\*sizeQ] neighborQ

4 arrays to represent previous and next of intersection points. [May be these arrays are not required]

int[sizeP\*sizeQ\*2] previousP  
int[sizeP\*sizeQ\*2] nextP  
int[sizeP\*sizeQ\*2] previousQ  
int[sizeP\*sizeQ\*2] nextQ

if intersection is empty and neighbors exist, then P1=Q1 (V-intersection)

# Phase 1: Intersection in GPU

Each thread in GPU considers one edge from P, and Q, and perform

If intersection is true, intersection points are saved to intersections arrays with alpha values.

neighbors arrays save the neighbors for each intersection point.

## Phase 2: Intersection Point Labeling

label array to save labels for each intersection point (IntersectionLabel and EntryExitLabel)

Steps

1. Input polygons P and Q.

2. Find edge intersections

# Dependencies in the Code

## Typedefs

intersectionType = {   
NO\_INTERSECTION, //0  
 X\_INTERSECTION, //1  
 T\_INTERSECTION\_Q, //2  
 T\_INTERSECTION\_P, //3  
 V\_INTERSECTION, //4  
 X\_OVERLAP, //5  
 T\_OVERLAP\_Q, //6  
 T\_OVERLAP\_P, //7  
 V\_OVERLAP //8 }

RelativePositionType = {  
 LEFT,  
 RIGHT,  
 IS\_P\_m,  
 IS\_P\_p }

## Function: intersect

Input: const edge& edgeP, const edge& edgeQ  
Output: double& alpha, double& beta

1. Decide which intersection type this is
2. Requires only involved edge P and edge Q
3. Calculates alpha (distance from p\_1 to intersection point) and beta (distance from q\_1 to intersection point) value
4. Call function A of point2D.h
5. Call fabs(returns absolute value) of cmath

## Function: computeIntersections

Input: polygon P and Q  
Output: intersection points, alpha/beta value, neighbor update

Iterate edges of P on SOURCE - edgeP  
 Iterate edges of Q on SOURCE - edgeQ  
 intersectionType = intersect(edgeP, edgeQ, alpha, beta)  
 P\_1=edgeP.one  
 Q\_1=edgeQ.one  
 case X\_INTERSECTION:  
 I= (1.0-alpha)\*edgeP.one->p + alpha\*edgeP.two->p  
 I\_P = new vertex(I,alpha);  
 I\_Q = new vertex(I,beta);  
 insertVertex(I\_P, edgeP);  
 insertVertex(I\_Q, edgeQ);  
 link(I\_P, I\_Q);  
 case X\_OVERLAP:  
 I\_Q = new vertex(P1->p, beta);  
 insertVertex(I\_Q, edgeQ);  
 link(P1, I\_Q);   
 I\_P = new vertex(Q1->p, alpha);  
 insertVertex(I\_P, edgeP);  
 link(I\_P, Q1);  
 case T\_INTERSECTION\_Q:  
 case T\_OVERLAP\_Q:   
 I\_Q = new vertex(P1->p, beta);  
 insertVertex(I\_Q, edgeQ);  
 link(P1, I\_Q);  
 case T\_INTERSECTION\_P:  
 case T\_OVERLAP\_P:   
 I\_P = new vertex(Q1->p, alpha);  
 insertVertex(I\_P, edgeP);  
 link(I\_P, Q1);  
 case V\_INTERSECTION:  
 case V\_OVERLAP:  
 link(P1,Q1);

## Function: vertex

Input: point2D=q, alpha\_value=a=-1.0  
Output: returns constructor of vertex (p=q, prev=next=neighbor=NULL, source=false, alpha=a, intersect={if alpha\_vlaues > default\_value})

## Function: link

Input: vertex=\*P, \*Q  
Output: intersection=True for P and Q,   
P->neighbor = Q  
Q->neighbor = P

## Function: oracle

Input: vertex \*Q, \*P1, \*P2, \*P3  
Output: RelativePositionType {LEFT, RIGHT, IS\_P\_m, IS\_P\_p}

// is Q linked to P1 ?  
if ( P1->intersection && (P1->neighbour == Q) )  
 return(IS\_P\_m);  
// is Q linked to P2 ?  
if ( P3->intersection && (P3->neighbour == Q) )  
 return(IS\_P\_p);  
// check relative position of Q with respect to chain (P1,P2,P3)  
double s1 = A( Q->p, P1->p, P2->p);  
double s2 = A( Q->p, P2->p, P3->p);  
double s3 = A(P1->p, P2->p, P3->p);  
if (s3 > 0) {   
 // chain makes a left turn  
 if (s1 > 0 && s2 > 0)  
 return(LEFT);  
 else  
 return(RIGHT);  
}else {  
 // chain makes a right turn (or is straight)  
 if (s1 < 0 && s2 < 0)  
 return(RIGHT);  
 else  
 return(LEFT);   
}

## Function: labelIntersection

Input: polygon P, Polygon Q, Intersection points updated  
Output: IntersectionLabel label, EntryExitLabel enex

// 1) initial classification  
Iterate edges of P on INTERSECTION=I  
 // determine local configuration at this intersection vertex  
 vertex\* P\_m = I->prev; // P-, predecessor of I on P  
 vertex\* P\_p = I->next; // P+, successor of I on P  
 vertex\* Q\_m = I->neighbour->prev; // Q-, predecessor of I on Q  
 vertex\* Q\_p = I->neighbour->next; // Q+, successor of I on P  
 // check positions of Q- and Q+ relative to (P-, I, P+)  
 RelativePositionType Q\_m\_type = oracle(Q\_m, P\_m, I, P\_p); dependency with other points not local  
 RelativePositionType Q\_p\_type = oracle(Q\_p, P\_m, I, P\_p); dependency with other points not local  
 //check non-overlapping cases  
 I->label={CROSSING, BOUNCING,}  
 //check overlapping cases  
 I->label={LEFT\_ON, RIGHT\_ON, ON\_ON, ON\_LEFT, ON\_RIGHT}

// 2) classify intersection chains  
Iterate edges of P on INTERSECTION=I  
 Use I->label from previous step   
 Get RelativePositionType x = {LEFT, RIGHT} (LEFT\_ON)  
 // proceed to end of intersection chain and mark all visited vertices as NONE   
 do { Need to traverse to == ON\_ON and change label of vertices while traverse   
 I->label = NONE;  
 I = I->next;  
 } while (I->label == ON\_ON);  
 Get RelativePositionType y = {LEFT, RIGHT} (ON\_LEFT)  
 chainType = {DELAYED\_CROSSING, DELAYED\_BOUNCING}  
  
// 3) copy labels from P to Q  
  
// 3.5) check for special cases  
choose P w.r.t Q or P w.r.t Q  
loop over all polygons in PP or QQ  
 if (P.noCrossingVertex(UNION)) {  
 // P\_ has no crossing vertex (but may have bounces or delayed bounces, except for UNION), hence it does not intersect with Q\_or\_P  
 noIntersection[i].insert(&P); // remember component, and ignore it later in step 4 **set data structure used**  
 // is P identical to some component of and Q\_or\_P?  
 if (P.allOnOn()) { this function traverse the whole polygon and check for ON\_ON   
 identical[i].insert(&P); // -> remember for further processing below  
 } else {   
 // is P inside Q\_or\_P?  
 bool isInside = false;  
 point2D p = P.getNonIntersectionPoint(); this function traverse the whole polygon and check for non-intersecting points  
 for (polygon& Q : \*Q\_or\_P)  
 if ( Q.pointInPoly(p) ) traverse polygons  
 isInside = !isInside;   
 if (isInside ^ UNION) {  
 RR.push\_back(P); // -> add P to the result  
 count[0]++;  
 }  
 }  
 }  
  
// handle components of P that are identical to some component of Q  
for (polygon\* P : identical[0]) { set filled form previous step  
 // is P a hole?  
 bool P\_isHole = false;  
 for (polygon& P\_ : PP)  
 if ( ( P\_.root != P->root ) && (P\_.pointInPoly(P->root->p)) )  
 P\_isHole = !P\_isHole;  
 for (polygon\* Q : identical[1]) set filled form previous step  
 for (vertex\* V : Q->vertices(ALL))  
 if (V == P->root->neighbour) { // found Q that matches P  
 // is Q a hole?  
 bool Q\_isHole = false;  
 for (polygon& Q\_ : QQ)  
 if ( ( Q\_.root != Q->root ) && (Q\_.pointInPoly(Q->root->p)) )  
 Q\_isHole = !Q\_isHole;  
 // if P and Q are both holes or both are not holes  
 if (P\_isHole == Q\_isHole) {  
 RR.push\_back(\*P); // -> add P to the result  
 count[1]++;  
 }   
 goto next\_P;  
 }  
 next\_P: ;  
 }

// 4) set entry/exit flags  
choose P w.r.t Q or P w.r.t Q (whole thing loops 2 times for each)  
loop over all polygons in PP or QQ  
// ignore selected polygon if it does not intersect with Q\_or\_P (detected in step 3.5 above)  
if(noIntersection[i].find(&P) != noIntersection[i].end()) continue;  
point\_in\_polygon test  
travers all intersection points starting from the one found before.   
 all operations are condition checking and independent assignments except folowing  
 if ( (I->label == BOUNCING) && ((status == EXIT) ^ UNION) )  
 split[i].insert(I); set data structure

// 5) handle split vertex pairs  
Loop all vertices of split[0]  
get neighbor of each vertex  
// check if the neighbor on Q is also a split candidate  
if (split[1].find(I\_Q) != split[1].end()) {  
 // compute areas to compare local orientation using prev and next values of current point and neighbor point  
 // link vertices correctly  
 if (sP\*sQ > 0) { // same local orientation  
 link(I\_P, V\_Q);  
 link(I\_Q, V\_P);   
 }else { // different local orientation  
 link(V\_P, V\_Q); }  
 // add duplicate vertices to P and Q   
 insertVertex(V\_P, I\_P);  
 insertVertex(V\_Q, I\_Q);  
 label enx of I\_P, V\_P, I\_Q, V\_Q complex dependency here

}

// 6) handle CROSSING vertex candidates  
loop all vertices of crossing[0]  
get neighbor of each vertex  
// check if the neighbor on Q is also a CROSSING candidate  
if (crossing[1].find(I\_Q) != crossing[1].end()) { label current and neighbor points}

## Function: createResult

Input: p of polygon vector PP, neighbor of each vertex  
Output:Resulting RR

loop CROSSING\_INTERSECTION vertices of each P  
 nested do{}while() to

# Notes

1. Same edge can have multiple intersection points. Need to position intersection points in the right order using alpha value (alpha increasing order for each edge?)
2. In computeIntersections(), linking can be only following
   1. P intersection point vs its q intersection counterpart
   2. Q/P\_1 with intersection point I\_P/Q
   3. P1 and Q1

# Parallel Random-Access Machine (PRAM) Polygon Clipping Algorithm with Ability to handle Degenerate Cases

This algorithm consists of 4 phases.

1. Intersection point calculation between two given edges of polygon P and Q in parallel
2. Insert intersection point in the correct position in parallel
3. Provide intersection type in partial parallel
4. Provide entry/exit label in parallel using point-in-polygon test

**Important**

1. For intersection, of each edge is static vertex and is considered an open vertex. becomes static in the next round when edge is evaluated.
2. Memory management
   1. if the polygons are large, but small enough to fit in single shared memory, we can complete step 1 in constant time/
   2. Each block/Streaming Multiprocessor (SM) works on a subset of the edge intersections using after copying data into shared memory. Each thread can learn its relative location to save the result in the global memory using the prefix sum of local K values as follows. (i) Constant time algorithm to determine number of intersection points (), (ii) calculate prefix sum of local K values. Time complexity for step 1 is in this scenario.

|  |
| --- |
| **Algorithm 1** Parallel Greiner-Hormann Polygon Clipping Including Degeneracy Handling |
| 1. Input: 2. Find edge intersection between . Insert to with ,neighbor (), ~~intersection type label, entry/exit label~~ 3. Each edge of input polygons generate an array of entries for polygon P and an array of entries for polygon Q along with count of intersection points for each polygon. 4. Sort intersection points by value of considering each edge. 5. For each intersection point, do a point-in-polygon test in parallel. 6. Create links between next, previous, and neighbor points using the calculated values (in shared memory). 7. Assign labels and labels in . |

# To Do

1. Create example polygon intersection following steps from the sequential algorithm
2. Complete PRAM algorithm. Write the easy steps we see at this point with some description.
3. Recommendations from advisors:
   1. check linked list ranking problem
   2. output data structure (2 input polygons with intersection points) is a graph. Draw a sample graph -> do a new graph traverse algorithm to generate output

|  |
| --- |
| **Algorithm 1 CREW** Greiner-Hormann Polygon Clipping Including Degeneracy Handling |
| 1. Input: 2. Find edge intersection between . Calculate values 3. Classify intersection type based on values {X-intersection, T-intersetion-P/Q, V-intersection, X-overlap, T\_overlap\_P/Q, V-overlap}. Insert vertex into relevant polygon (P or/and Q), link with relevant neighbor . ( V-intersection/overlap) 4. Labeling phase consists of 6 sequential stages. Calculate . Calculate relative position type of based on above mentioned parameters and intersection point.    1. Classify relative position type {LEFT, RIGHT}\_TURN, IS\_P\_{m, p}. Classify non-overlapping cases {CROSSING, BOUNCING}, and overlapping cases {RIGHT\_ON, LEFT\_ON, ON\_ON, ON\_LEFT, ON\_RIGHT} label based on relative position type.    2. Intersection vertices of with adjacent overlapping edges form polygonal intersection chains , where is marked as are marked as and is marked as with . Classify intersection chains {DELAYED\_CROSSING, DELAYED\_BOUNCING} using overlapping case from step (a).    3. Special case handling. Loop over components of polygon P and Q. Uses noCrossingVertices(), allonon(), getNonIntersectionPoint()    4. Apply entry/exit label. Loop over P and Q. If P and Q intersects, getNonIntersectionVertex(). Label all intersections using point-in-polygon test.    5. Handle split vertex pairs.    6. Handle CROSSING vertex candidates 5. Sequential trace result. |

# Latest GH algorithm – Example

|  |  |  |
| --- | --- | --- |
| P | x | y |
| 1 | 225 | 550 |
| 2 | 775 | 500 |
| 3 | 725 | 200 |
| 4 | 625 | 350 |
| 5 | 425 | 175 |

|  |  |  |
| --- | --- | --- |
| Q | x | y |
| 1 | 525 | 525 |
| 2 | 625 | 350 |
| 3 | 300 | 225 |
| 4 | 150 | 425 |

A picture containing diagram

Description automatically generated

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R | x | y | α | β |  |
| 1 | 525 | 525 |  |  |  |
| 2 | 625 | 350 |  |  |  |
| 3 | 385 | 258 |  |  |  |
| 4 | 278 | 459 |  |  |  |

3, 725, 200

1, 225, 550

2, 775, 500

4, 625, 350

5, 425, 175

1, 500, 525

2, 625, 350

3, 300, 225

4, 150, 425

1, 500, 525

3, 385, 258

4, 278, 459

3, 385, 258

4, 278, 459

500, 525  
Neither  
Bounce

625, 350  
Neither  
Bounce

300, 225

382, 256  
Exit  
Cross

150, 425

273, 460  
Entry  
Cross

225, 550

500, 525  
Neither  
Bounce

775, 500

725, 200

625, 350  
Neither  
Bounce

425, 175

382, 256  
Entry  
Cross

273, 460  
Exit  
Cross

**GPU Algorithm 1: Steps**

1. Copy and to GPU. Assume
2. Each thread calculates edge with all edges of . Calculate count of intersection points and intersection point count excluding degenerate cases . Compute prefix sum of and prefix sum of with extra 1 (to represent ) added in each step .
3. Similarly, calculated. 2 options to complete this step.
   1. duplicate intersection calculation each edge intersects by all edges of . Time complexity is ,
   2. no duplicate calculations. Follow reduction tree to reduce prefix sums of each thread (reduce local prefix sum arrays to one array). Time complexity is
4. Insert intersection point with in in the correct location using start point and local index incremented in the local intersection calculation. The first vertex of edge, is also appended in before inserting intersection points.
5. Similarly, is created using . Reduce arrays to compute final (merge intersection points in each edge and sort them using ). Time complexity for reduction is .
6. Array is used to track parent vertices of each intersection point from and . Initially, each thread has its own . All of them are reduced to get the final . Only may change in the process of sorting and reduction. When each thread finds an intersection point, it saves (index of intersection point in is ). helps to read neighbors of in constant time.
7. Sort intersection points in each edge of using .Local indices are updated in . Time complexity for sorting is using quicksort if any edge intersects with all edges of .
8. arrays create the above graph.
9. Perform initial classification using the neighbor and predecessor, successor information in parallel.
10. Classify intersection chains in sequential.
11. For special case calculations, need following data structures which are saved in parallel.
    1. Non intersection vertices list of P and Q (noIntersection list)
    2. List of vertices with ON\_ON label. This need to complete step 10 which is sequentially done (for both P and Q) (identical list)
    3. Lists of point-in-polygon tests with respect to both P and Q of non-intersection vertices. Will perform in parallel. To test a given vertex is inside a polygon of size n, can be accomplished in constant time using n number of processors.
12. Step 3.5 to 6 in sequential using data structures from above

Important

1. It is not required to use consequent threads among each other. Instead, we need to use shared memory effectively. The roadmap to optimize the code is by using the shared memory for local computations.
   1. Each SM has a shared memory of 48KB=49152B.
   2. Each SM has 8 blocks.
   3. If we fit computation into a single block, each block should hold,

size m input XY double + intersection XYAlpha double + input XY double = m\*sizeof(double)\*2 + k\*sizeof(double)\*3 + sizeof(double)\*2 =

m\*8\*2 + k\*8\*3 + 8\*2 = (16m+24k+16)B

in worst case, m=k; 40m+16=49152 -> m=1228

if shared memory is only for input data m=3071

If we want to keep the size as a multiple of 32 for optimal warp occupancy, m=3040  
;where sizep=n, sizeQ=n, #intersections found=k

* 1. If the shared memory is not enough, then need to use multiple blocks to fit the data. There are 2 configurations in this case,
     1. Use single SM to fit data – Still too small. Can be done by copying batches of input data and then perform intersection batch wise. Could be complex and might need warp level synchronization?
     2. Use multiple SMs (multiple shared memory blocks) – Still hard to fit larger polygon into all shared memory together.
  2. Idea is to use a hybrid of both ideas and it may also maintain higher warp occupancy.
  3. Also, need warp occupancy helps to optimize the code. At a time, we need to use all SMs throughout the GPGPU processing.
     1. #threads=threadsPerBlock=256 (range for this value is 128-512 with multipliers of 32)
     2. #blocks=(m+n)/ threadsPerBlock

Thread method()

Loop for all tiles()  
 Load partial data tile into shared memory   
 Synchronize threads  
 perform intersection in loop()  
 synchronize threads

End loop

End method