Analyzing the Performance of GPU-Based Point-in-Polygon Algorithms

Anish Kambhampati
Department of Computer Science
Purdue University
West Lafayette, USA
akambha@purdue.edu

***Abstract*—The point-in-polygon (PIP) query determines whether a point lies inside or outside a polygon. It is a fundamental problem in computational geometry, with applications spanning computer graphics, geographic information systems, and scientific simulations. While traditional approaches to this query, such as the ray casting algorithm and winding number algorithm, rely on CPU-based computations, advancements in GPU computing have enabled high-performance alternatives. In this paper, we benchmark three GPU-based PIP algorithms: (1) a straightforward CUDA-based ray-casting implementation, (2) RayJoin, a state-of-the-art algorithm leveraging NVIDIA's hardware-accelerated ray tracing (RT), and (3) s-ray, a novel approach using NVIDIA's RT hardware with a per-edge triangle index. We evaluate these methods against two distinct scenarios: a high point-to-polygon ratio and a more balanced point-to-polygon ratio.**

Our findings reveal significant trade-offs between query performance and build (preprocessing) costs. While the CUDA-based implementation is simple and build-free, it is the slowest in query throughput. RayJoin achieves up to 30x speedups over the CUDA algorithm by optimizing edge groupings into axis-aligned bounding boxes (AABBs). The s-ray algorithm further doubles RayJoin's throughput, albeit at the cost of 6–20x longer build times. Our work highlights the tradeoffs between different GPU-based PIP algorithms, providing a foundation for users to select and/or develop their own high-performance PIP algorithms.

***Index Terms*—point-in-polygon(pip), ray tracing, gpu, cuda**

I. Introduction

Spatial queries have a wide variety of applications, including in geospatial systems, computer graphics, and more. Examples of spatial queries include computing the overlap between two polygons, computing the distance between two geometries, and (most importantly for our research) determining whether a point lies within a given polygon. This last spatial query is knows as the point-in-polygon test and has been the subject of much research.

Traditional implementations of the point-in-polygon use the CPU for processing. One such implementation is the ray casting algorithm, which involves shooting a ray from the point in any direction and counting the number of intersections that ray makes with the polygon [3]. An even number indicates that the point is outside the polygon; an odd number indicates that the point is inside the polygon. Another implementation involves computing the *winding number* of the point with respect to the polygon. Then, the point is inside the polygon if and only if its winding number is nonzero [2].

Recent advances in GPU computing, however, allow us to develop algorithms that run on the GPU. These algorithms can take advantage of features such as massively parallel processing and hardware-accelerated ray tracing. In this paper, we benchmark and compare three such GPU-based algorithms. The first is a straightforward CUDA-based implementation of the ray casting algorithm. Using CUDA allows the algorithm to compute the PIP query for many points in parallel, leading to dramatic speed-ups over CPU-based implementations. The second algorithm is RayJoin, a state-of-the-art spatial join algorithm that leverages NVIDIA's hardware-accelerated ray tracing (RT) capabilities [1]. The third algorithm is s-ray, a work-in-progress novel algorithm that also uses hardware-accelerated RT capabilities. Unlike RayJoin, however, s-ray uses a per-triangle index for its build step. RayJoin, on the other hand, uses edge groupings in its build step. To test these algorithms, we evaluate them on two distinct scenarios. The first scenario has an order of magnitude more polygons than points. The second scenario involves a more balanced ratio of polygons and points.

Upon testing, we find that all three algorithms have varying build (preprocessing) costs and query costs. The CUDA approach, due to its simplistic algorithm, does not involve a build step at all, but its query time is consequently the slowest among the three approaches. RayJoin has significantly higher query throughput but this comes at the cost of introducing mild build costs. s-ray is the fastest of the three algorithms in both scenarios, but its build costs are an order of magnitude higher than RayJoin's.

We show that GPU computing, and NVIDIA RT architecture in particular, holds enormous potential in revolutionizing spatial database systems. We believe that our benchmarking work is a testament to this potential, and our contributions in this paper are as follows:

- We have written a CUDA-based ray casting implementation of the point-in-polygon query
- We have cleaned and prepared novel datasets for use in our benchmarking that simulate scenarios with significantly more points than polygons and scenarios with a more balanced set of points and polygons.
- We have performed the benchmarking of the three aforementioned algorithms across these two distinct scenarios.
- We have made a strong case as to the potential of ray

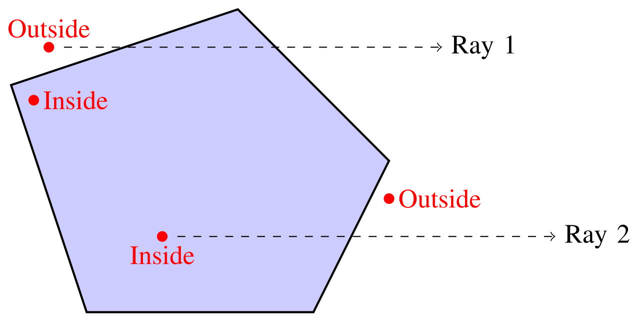

Fig. 1. An example of the point-in-polygon (PIP) query using the ray casting method. The polygon is shown in blue, with points labeled as inside or outside. An even number of ray-polygon intersection (ray 1) indicates that the point is outside the polygon; an odd number (ray 2) indicates the opposite.

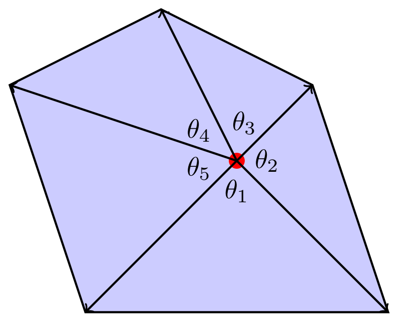

Fig. 2. An example of the point-in-polygon (PIP) query using the winding number algorithm. By adding up the signed angles subtended by each polygon edge at the point in question (θ_1 through θ_5), we can verify that the winding number is zero. This means that the point is *inside* the polygon, as is visually apparant. Note that this algorithm is not just restricted to convex polygons as shown in the figure.

tracing-based algorithms, as evidenced by the superiority of RayJoin and s-ray over the CUDA approach.

II. Background

A. Existing Algorithms

The point-in-polygon (PIP) problem, at its core, seeks to determine whether a given point lies in a given polygon. Historically, early solutions to PIP queries relied on simple geometric principles. One example would be the ray casting algorithm, which is shown by Figure 1. In this approach, a ray is drawn from every point being testing out to infinity. The number of ray-polygon intersections is computed and its odd-even parity is used to determine the answer to the query. This ray can be used to test whether the point in question intersects any number of candidate polygons. In practice, CPU-based algorithms do not actually draw the ray due to computational constraints. Instead, they consider each edge of the polygon, check whether the ray would intersect that edge using mathematical formulas, and invert a boolean counter if so. The exact approach is discussed more with the CUDA algorithm.

Another such algorithm would be the winding number algorithm. Unlike the ray-casting algorithm, which counts intersections of a ray with polygon edges, the winding number algorithm computes the number of times the polygon winds around the point in question. This is achieved by summing up the signed angles subtended by each edge of the polygon at the point. If the total winding number is zero, the point is outside the polygon; otherwise, it is inside [2]. The algorithm is particularly well-suited for complex polygons, including those with self-intersections, as it inherently accounts for the orientation of edges. It is, however, more computationally intensive than ray-casting, due to expensive trigonometric functions involved in its calculations.

B. GPU Computing

Unlike traditional CPU-based computing, GPU-based computing leverages the parallel processing capabilities of GPUs to accelerate computations. These capabilities come from the fact that while CPU have relatively few powerful cores, GPUs have thousands of lightweight cores designed to execute many tasks simultaneously [7]. NVIDIA's CUDA is a widely adopted platform and programming model for general-purpose GPU computing, which allows developers to write high-performant parallel code in languages like C++ and Python [4]. CUDA enables developers to control low-level GPU features. This sort of control is useful when writing highly optimized code as certain parallelizable sections can be offloaded to the GPU for computation.

Another advanced feature of GPUs is their hardware support for ray tracing. At its core, ray tracing simulates the behavior of light by tracing rays as they interact with objects in a scene, providing highly accurate visuals for applications such as computer graphics, simulations, and computational geometry. While often used for video games and 3D visualizations, the same technology does have applications in high-performance computing (HPC). This is made possible by NVIDIA's OptiX, a ray-tracing engine built on CUDA that provides a programmable interface for ray-tracing tasks [8]. OptiX leverages dedicated hardware, such as the RT cores in NVIDIA GPUs, to accelerate operations like ray-object intersection tests and bounding volume hierarchy (BVH) traversals. These capabilities can be exploited by spatial queries which often have the same intersection test requirements.

C. Bounding Volume Hierarchy

A bounding volume hierarchy (BVH) is a spatial data structure commonly used in computer graphics to optimize the ray tracing process. It organizes geometric primitives, such as triangles, into a tree-like structure of bounding volumes [6]. Each node in the BVH represents a bounding volume, typically an axis-aligned bounding box (AABB), that encapsulates a subset of the scene's geometry. The root node encompasses the entire scene, while the leaf nodes contain the actual primitives. Fig. 3 shows an example of a BVH. By hierarchically subdividing the scene, these hierarchies enable efficient culling of objects that do not intersect with a given ray. During traversal, only the nodes intersected by the ray are processed, significantly reducing the number of intersection tests required and accelerating rendering performance.

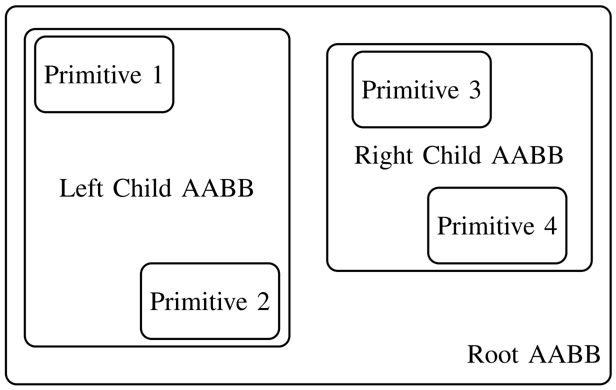

Fig. 3. An example of a bounding volume hierarchy (BVH). Spatially adjacent geometries are grouped together in a way that makes intersection traversal significantly faster.

In ray tracing, the BVH is crucial for balancing memory usage and computational efficiency. Construction of a BVH involves organizing primitives to minimize the overlap between bounding volumes and optimize traversal cost. The quality of the hierarchy impacts the rendering speed, as a well-constructed BVH reduces the number of intersection tests during ray traversal. In HPC applications, BVH constructions are used to carefully tune build and query times. A more precise BVH will result in faster query times, but at the cost of a large build time. The opposite is true of a coarse BVH, or one that wraps multiple geometries in a single leaf node.

III. Algorithms

Having given a rough overview of the concepts involved, we now dive deeper into the technical underpinnings behind each algorithm being benchmarked.

A. CUDA-based Ray Casting

The CUDA-based ray casting approaches largely follows the standard ray casting implementation. The only difference is that ray casting is done for each point in parallel using CUDA. Each polygon, however, is checked serially for each point. Note that the code only returns the first polygon that the point lies in. This assumption is in the majority of cases valid, as for example, if we want to determine which American state a point lies in, there will not be two states that the point lies in. Put another way, it is not often in the real world that geographic regions overlap. The high-level pseudocode is shown in algorithm 1. The `CHECKPOINT` function on line 8 is called in parallel for each point, and each function invocation calls the `CHECKPOLYGON` function on line 1 serially. Note that the pseudocode is an abstraction of the CUDA programming involved; rather than return values directly, we instead read and write to blocks of memory that are then transferred between the GPUs and host CPUs.

Algorithm 2 provides a more technical explanation of how the ray casting approach is implemented. More specifically, it showcases the computing logic behind the `CHECKPOLYGON`

Algorithm 1 CUDA-Based Point in Polygon Query

```
1: procedure CHECKPOLYGON($pt, P$)                          ▷ Device
2:     $c \leftarrow$ false
3:     for each edge $e \in P$ do
4:         if ray from $pt$ intersects $e$ then
5:             $c \leftarrow !c$
6:     return $c$                                      ▷ Whether $pt \in P$
7:
8: procedure CHECKPOINT($pt, polygons$)                     ▷ Kernel
9:     for each polygon $P \in polygons$ do
10:        if CHECKPOLYGON($pt, P$) then
11:            return $P$
12:    return -1                                      ▷ No polygon found
```

Algorithm 2 Ray-Casting Algorithm for PIP

```
1: Input: $xp$, $yp$ (arrays of x and y coordinates of the polygon vertices), $npol$ (number of vertices), $x, y$ (coordinates of the query point)
2: Output: c (boolean indicating whether the point is inside the polygon)

3: $j \leftarrow npol - 1$, $c \leftarrow$ false
4: for $i = 0$ to $npol - 1$ do
5:                      ▷ $i$ is the current vertex, $j$ is the previous vertex
6:     $b \leftarrow ((yp[i] \leq y) \land (y < yp[j])) \lor ((yp[j] \leq y) \land (y < yp[i]))$     ▷ Whether $y$ is between edge y-values
7:     $xd \leftarrow (xp[j] - xp[i]) \cdot (y - yp[i])/(yp[j] - yp[i]) + xp[i]))$     ▷ x-coordinate of ray-edge intersection

8:     if $b \land x < xd$ then
9:         ▷ $y$ is in between and $x$ is to the left of ray-edge intersection
10:        $c \leftarrow !c$                              ▷ ray-edge intersection
11:
12: return c                                          ▷ Return the result
```

function in algorithm 1. A technical overview of the ray casting logic is as follows:

- The ray-casting algorithm determines whether a point lies inside a polygon by drawing a ray from the point and counting its intersections with the polygon's edges.
- It iterates through each edge of the polygon, checking whether the y-coordinate of the query point lies between the y-coordinates of the edge's endpoints.
- For edges that intersect the ray, it calculates the x-coordinate of the intersection and compares it to the query point's x-coordinate.
- If the query point's x-coordinate is less than the x-coordinate of the intersection, the algorithm toggles a boolean flag.
- At the end of the loop, the boolean flag indicates whether the point is inside (`true`) or outside (`false`) the polygon.
- The algorithm handles edge cases, such as points lying

exactly on edges, by ensuring proper comparison logic, i.e. one use of $\leq$ and one use of $<$ in the y-coordinate comparisons.

B. RayJoin

RayJoin is a SOTA spatial join library that leverages NVIDIA's RT Cores to overcome the performance limitations of conventional spatial join methods on both CPUs and GPUs. RT Cores, designed for hardware-accelerated Bounding Volume Hierarchy (BVH) traversal in ray tracing, enable RayJoin to execute spatial queries like line segment intersection (LSI) and PIP tests efficiently (though we are specifically interested in the PIP tests). By reformulating PIP as a ray tracing problem, RayJoin achieves significant throughput gains compared to traditional CPU-based implementations [1].

Fundamentally, RayJoin works by shooting a ray from every single point being queried and checking the closest polygon that the ray hit. This closest polygon must be the polygon that the point was located in. On a CPU, this sort of ray casting would be no better than the algorithm used for the CUDA implementation. But with NVIDIA's hardware RT acceleration, this approach can be significantly faster. There are some challenges associated with using RT, however.

One issue is having to construct a BVH around each edge. To avoid high build costs associated with having each edge of every polygon be its own primitive, RayJoin uses an Adaptive Grouping (AG) technique. This addresses the high time and memory costs associated with constructing BVH trees by grouping spatially close line segments into shared Axis-Aligned Bounding Boxes (AABBs). Instead of creating a unique AABB for each line segment, AG merges adjacent AABBs based on an area expansion ratio threshold, ensuring minimal dead space while reducing the total number of primitives. This approach allows efficient representation of grouped line segments as ranges, facilitating quick linear scans during ray traversal. To implement AG on GPUs, line segments are partitioned into groups processed in parallel by thread blocks, with each thread attempting to merge adjacent AABBs within its partition. If the area expansion ratio between two AABBs meets the threshold, they are merged; otherwise, they remain separate. This iterative process continues until no further merges are possible.

Another problem that RayJoin solves is the precision issue that comes from OptiX's usage of FP32. While fine for video games (and much faster to work with than FP64 numbers), not using FP64 numbers creates precision issues that are unacceptable for spatial database calculations. RayJoin addresses these challenges by using a technique called Conservative Representation (CR). In FP32 hardware, downcasting from higher precision, causes geometric inconsistencies due to the truncation of mantissa bits. CR ensures that the downcasted bounding boxes (AABBs) are large enough to fully enclose the high-precision line segments. This is achieved by tweaking the mantissa bits of the boundaries of the AABBs to ensure they properly enclose the line segments after downcasting, without excessively enlarging the AABBs, which could lead to false positives.

These optimizations ensure that RayJoin remains a SOTA implementation of the PIP query. Previous testing has revealed that RayJoin performs between 3 and 28 times faster than any other existing methods [1].

C. s-ray

S-ray is similar to RayJoin in the sense that both use NVIDIA RT cores to accelerate PIP computations. The main difference is how the build indices are constructed. In RayJoin, these indices are constructed as AABBs around the edges of the polygons. As discussed earlier, multiple edges may be grouped into the same AABB to save on build time. In s-ray, however, the indices are instead created as triangles. Triangles are used because the RT cores have native support for this geometry. Given a polygon, one triangle is created for each edge. If we denote the endpoints of the edge as $(x_i, y_i, 0)$ and $(x_j, y_j, 0)$, then the vertices of the corresponding triangle would be $(x_i, y_i, 0)$, (x_j, y_j, ε), and $(x_j, y_j, -\varepsilon)$, where $\varepsilon = 10^{-16}$. Now, we shoot a ray from every point in the horizontal x direction. Then, the RT cores will record an intersection between the ray and some triangle that we created. From the triangle id, we can deduce the edge, and consequently the polygon, that the point lies in. Algorithm 3 provides a lower-level overview of the device code involved in performing the ray tracing. S-ray forms the final PIP implementation that we will benchmark.

Algorithm 3 CUDA-Based Point in Polygon Query Using Ray Casting

```
procedure ANYHITTRIANGLEMESH
    $primID \leftarrow$ GETPRIMITIVEINDEX()
    $qID \leftarrow$ GETLAUNCHINDEX().x
    RESULTBUFFER[$qID$] $\leftarrow primID$
    IGNOREINTERSECTION()

procedure OPTIX_ANY_HIT_PROGRAM
    ANYHITTRIANGLEMESH()

procedure OPTIX_CLOSEST_HIT_PROGRAM
    $primID \leftarrow$ GETPRIMITIVEINDEX()
    $qID \leftarrow$ GETLAUNCHINDEX().x
    RESULTBUFFER[$qID$] $\leftarrow primID$

procedure OPTIX_RAYGEN_PROGRAM
    $qID \leftarrow$ GETLAUNCHINDEX().x
    RESULTBUFFER[$qID$] $\leftarrow$ 4294967295        ▷ Default
    $data \leftarrow$ GETPROGRAMDATA()
    $origin \leftarrow$ RAYVERTICES[$qID$]
    $direction \leftarrow \langle 1, 0, 0 \rangle$        ▷ Shoot in +x direction
    $tmax \leftarrow$ FLOAT_MAX
    $ray \leftarrow$ CREATERAY($origin$, $direction$, 0, $tmax$)
    $color \leftarrow 0$        ▷ Value not used
    TRACERAY($data.world$, $ray$, $color$)
```

TABLE I
NUMBER OF POINTS FOR SYNTHETICS POLYGON SETS FOR UNIFORM & GAUSSIAN POLYGON DISTRIBUTIONS

Polygon Count	Number of Points (Uniform Distribution)	Number of Points (Gaussian Distribution)
1M	8,042,719	8,372,516
2M	16,185,787	17,533,798
3M	24,420,288	27,527,640
4M	32,754,122	38,411,735
5M	41,175,683	50,191,918

IV. EVALUATION

A. Datasets

To accurately test the overall performance of each algorithm, we benchmark all three in two distinct scenarios. In the first scenario, there are significantly more points than polygons. To cover this scenario, we use 8 months of NYC taxi trip start point data as the points and the NYC borough boundaries as the polygons. After removing trips without start point data, the resulting dataset consists of 111,041,607 points. An important thing to note is that while the CUDA baseline takes in a shapefile for the polygons and a custom binary to store the points, both RayJoin and s-ray use the CDB format for both inputs. (The precise specifications of the CDB format is described in [5].) We therefore had to convert the list of points to the CDB format. As for the polygons, there are five boroughs in NYC; however, each borough cannot be perfectly represented by a single polygon. We used ArcGIS to provide an authoritative map of the boroughs, which gave a total of 117 polygons with 82,728 vertices.

The second scenario involves a more balanced distribution of points and polygons. For this scenario, we used RayJoin's synthetic polygon generation scripts to generate sets of polygons of size 1 million (1M) through 5 million (5M), in both uniform and Gaussian distributions. The vertices of each polygon, then, become the points used in the PIP query. The precise number of points per polygon per distribution is shown in table I. All tests were conducted with 5M polygons. Once the polygons were generated, we used ArcGIS to convert them to the required CDB format. We did not convert these to the custom binary format required for the CUDA implementation as we did not feel CUDA would perform reasonably well in this scenario. This is because while it parallelizes the checking of every point, each point must be serially checked against each polygon. This is extremely slow, and we concluded that this issue makes the CUDA ray casting approach infeasible in situations with a large number of polygons.

B. Results

The first set of results we examine is the case with many points and few polygons. Fig. 4 shows the results. From this, it should be clear that the CUDA ray casting implementation has by far the slowest query time, being over 28x slower than RayJoin. It does, however, have the benefit of not requiring a build step. RayJoin does require such a step, but at only

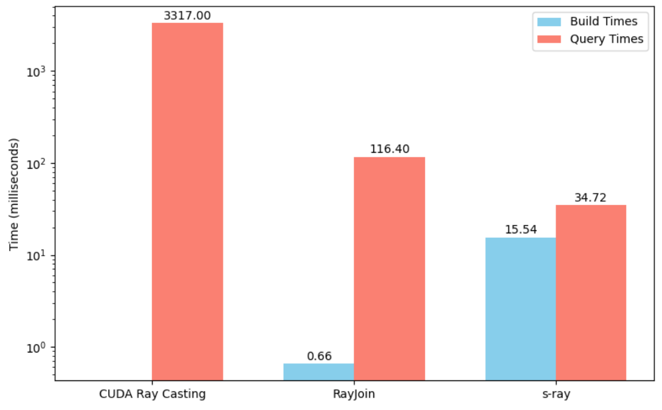

Fig. 4. Build and query times on the NYC taxi dataset for CUDA ray casting, RayJoin, and s-ray.

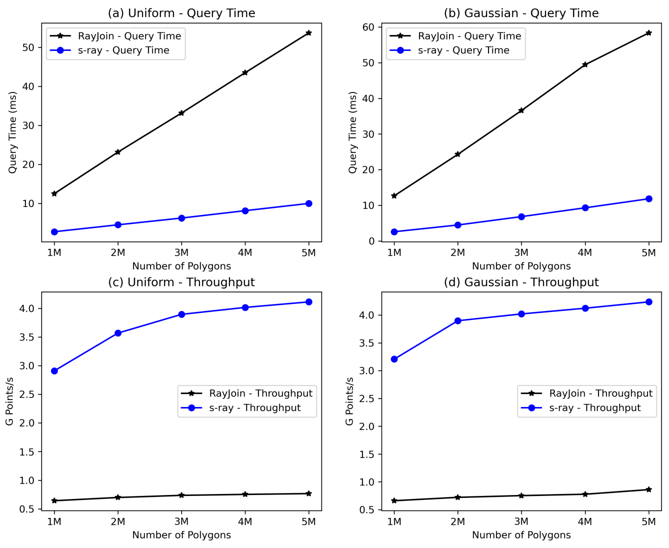

Fig. 5. Query time and throughput as a function of the number of polygons used, RTX 4070 Ti.

0.66ms, it is largely negligible. Most remarkably, the novel s-ray algorithm achieves a 3.35x speedup over RayJoin. Though its build time is much slower than RayJoin's (due to not implementing some form of adaptive grouping), the combined build + query time is still over twice as fast.

The second set of results is for the case with a more balanced number of points (between 8-50M) and polygons (5M). We benchmark s-ray and RayJoin for this case. (As discussed earlier, practical limitations make testing the CUDA vesion unnecessary.) We run each query with 5 "warm-up" rounds (which are not considered in the final time) and 5 actual rounds, whose average build & query times are recorded. Furthermore, we run each query on two different systems to gain a better insight into performance. The first system (hereafter referred to as "RTX 4070 Ti") contains two NVIDIA GeForce RTX 4070 Ti graphics cards. The second system (hereafter referred to as "RTX 6000") contains two NVIDIA RTX 6000 Ada Generation graphics cards. These cards are

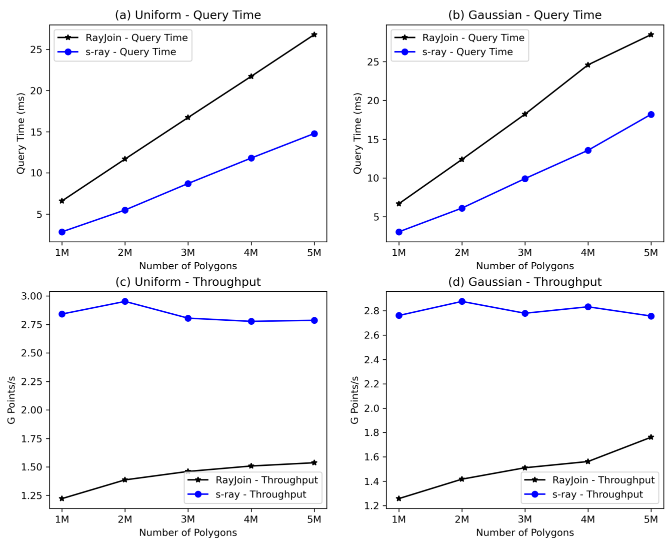

Fig. 6. Query time and throughput as a function of the number of polygons used, RTX 6000.

significantly more powerful than the 4070 Ti cards. The results of running these tests are displayed in Table II. From the data, it is clear that while RayJoin has a significantly faster build time, s-ray has the faster query time.

To take a closer look at how query time changes with the number of points, we construct two sets of graphs (one for the RTX 4070 Ti and one for the RTX 6000). Each set contains graphs showing how query time and throughput (defined as query points divided by the query time) change with the number of polygons used for the points. The RTX 4070 Ti graphs are shown in Fig. 5; the RTX 6000 graphs are shown in Fig. 6.

From the graphs, we can make a couple of observations. Firstly, the query times of RayJoin and s-ray appear to scale linearly with the number of polygons. This is a good indication that both solutions can adequately scale to larger datasets. The choice of polygon distribution appears to have little effect on this trend. The more interesting graphs, in our opinion, are the bottom two of each graph set. These showcase the throughput of each implementation. Unsurprisingly, the faster s-ray has a high throughput. There are some interesting observations about this though. For one, whereas the s-ray graph appears fairly consistent in its throughput regardless of polygon count, RayJoin appears to have a slight increase in throughput with more points. This finding is consistent with what the authors of that implementation found [1]. The other major observation to make is that s-ray's throughput is higher on the RTX 4070 Ti than the RTX 6000. This is not the case with RayJoin.

V. Discussion

A. *CUDA Ray Casting vs RayJoin vs s-ray*

From the first experiment, we can see that CUDA ray casting has the lowest throughput of the three algorithms. The main reason for this has to do with optimization. Ray-tracing cores are specifically designed for efficient intersection tests, using dedicated hardware acceleration to process rays in a highly optimized manner. In contrast, the CUDA ray casting algorithm relies on general-purpose streaming multiprocessors (SMs), which are less specialized and must execute intersection calculations in software. Another point to consider is that RT cores can exploit optimized data structures like BVHs in a way that CUDA-based approaches simply cannot.

Across both experiments, we can see that while s-ray has the faster query time, RayJoin has the faster build time. As mentioned previously, this stems from RayJoin's adaptive grouping (AG) step, which decreases the number of primitives needed in the BVH. In the first experiment, with many points and few polygons, the combined build + query time for s-ray was faster than that of RayJoin. In the second experiment, however, the opposite was true. This combination indicates that both algorithms may have their use in HPC applications.

B. *Uniform vs Gaussian Distributions*

From Table II Gaussian distributions result in build and query times that are approximately 3-4% larger than their uniformly distributed counterparts. This is true for both RayJoin and s-ray, as well as for both the RTX 4070 Ti and RTX 6000. The reason behind this has to do with spatial locality and the efficiency of RT acceleration structures, such as bounding volume hierarchies (BVHs). Uniformly distributed points are more evenly scattered, allowing BVHs to process queries more predictably with balanced traversal costs. In contrast, Gaussian-distributed points cluster around the mean, leading to uneven workloads as certain regions of the BVH experience higher query densities. This clustering increases traversal overhead and memory access contention, resulting in the observed build and query time difference.

C. *RTX 4070 Ti vs RTX 6000*

Unsurprisingly, all build times were slower on the RTX 6000 than the RTX 4070 Ti. The difference can be attributed to the power of the graphics card. The RTX 6000 is a high-end workstation GPU optimized for professional workloads, including ray tracing, with higher computational throughput, more dedicated ray tracing cores, and larger memory capacity. In contrast, the RTX 4070 Ti, while powerful, is designed primarily for consumer-grade applications like gaming, with fewer resources allocated to professional-grade ray tracing workloads. Query times, however, are a little more interesting. As mentioned previously, while RayJoin does exhibit slower query times on the 4070 Ti, s-ray does not. In fact, it exhibits the opposite behavior – throughput is in fact higher on the 4070 Ti. This combination of higher build and faster query times on the 4070 Ti is quite peculiar, and the reasons behind this are still not fully understood. We leave it to future work to explore this discrepancy.

VI. Future Work

A. *Adaptive Grouping for s-ray*

S-ray is a work in progress, and there are some ideas that could help increase its performance. One such step could be

TABLE II
QUERY TIME (BUILD TIME) IN MILLISECONDS FOR PIP QUERIES ON UNIFORM AND GAUSSIAN POINT DISTRIBUTIONS ACROSS TWO NVIDIA GPUS

		RayJoin		s-ray	
		RTX 4070 Ti	RTX 6000	RTX 4070 Ti	RTX 6000
Uniform	1M	12.514 (46.152)	6.583 (27.961)	2.764 (143.331)	2.830 (92.573)
	2M	23.156 (46.109)	11.677 (27.949)	4.536 (142.364)	5.482 (92.241)
	3M	33.163 (46.059)	16.722 (27.922)	6.269 (142.941)	8.705 (91.789)
	4M	43.507 (45.843)	21.723 (27.966)	8.156 (142.296)	11.793 (92.380)
	5M	53.695 (46.030)	26.798 (27.993)	10.013 (143.015)	14.777 (92.345)
Gaussian	1M	12.671 (49.146)	6.660 (29.766)	2.610 (148.723)	3.032 (96.556)
	2M	24.302 (49.113)	12.376 (29.566)	4.497 (148.530)	6.095 (96.225)
	3M	36.583 (48.996)	18.224 (29.655)	6.845 (148.395)	9.904 (96.237)
	4M	49.417 (49.041)	24.597 (29.504)	9.314 (149.160)	13.560 (95.728)
	5M	58.339 (45.035)	28.500 (26.446)	11.843 (148.985)	18.207 (96.525)

the implementation of an adaptive grouping (AG) step to tame the high build costs, in a similar vein to RayJoin. This would, of course, increase the query costs, as there would be more work involved in the ray tracing intersection shaders to see which primitive (if any) was actually hit. The key would be to see if AG could sufficiently decrease build costs without overly increasing the query costs.

B. Polygon Parallelization for CUDA Ray Casting

The current implementation of CUDA ray casting involves checking every polygon serially, even though the points themselves are checked in a parallel fashion. One optimization would be to see if the polygons and points can both be checked in parallel. This could introduce memory contention errors if two GPU threads try to write intersections about a point to the same location in memory. However, if fully realized, it could be possible to run the same synthetic benchmarks that were used with RayJoin and s-ray. This could also speed up the CUDA ray casting approach, though it is unlikely that it would ever achieve a throughput as high as RT-based approaches for reasons discussed earlier.

VII. CONCLUSION

In this paper, we explored three different GPU-based algorithms for PIP computations. We analyzed their performance across two distinct scenarios and contrasted the strengths of the three algorithms. This is extremely relevant to end users who need to select the most performant algorithm for their workloads. We have also provided possible future directions for the algorithms, which can aid researchers in developing even stronger PIP implementations. In both cases, our hope is that this paper serves as a strong foundation for the future of GPU-based PIP queries.

ACKNOWLEDGMENTS

We would like to thank Durga Mandarapu, a current PhD student in computer science at Purdue University, and Dr. Milind Kulkarni, the Michael and Katherine Birck Head of ECE and Professor, for their valuable mentorship. We also thank Dr. Zahra Tehrani for her assistance in running the Undergraduate Research Fellowship (URF), without which this research would not have been possible.

REFERENCES

[1] L. Geng, R. Lee, and X. Zhang, "RayJoin: Fast and Precise Spatial Join," in Proceedings of the 38th ACM International Conference on Supercomputing, 2024, pp. 124–136. doi: 10.1145/3650200.3656610.
[2] K. Hormann and A. Agathos, "The point in polygon problem for arbitrary polygons," Computational Geometry, vol. 20, no. 3, pp. 131–144, 2001, doi: https://doi.org/10.1016/S0925-7721(01)00012-8.
[3] D. Horvat and B. Zalik, "Ray-casting point-in-polyhedron test," Proceedings of the CESCG, 2012.
[4] D. Kirk and others, "NVIDIA CUDA software and GPU parallel computing architecture," in ISMM, 2007, vol. 7, pp. 103–104.
[5] S. V. G. Magalhães, M. V. A. 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. doi: 10.1145/2835185.2835188.
[6] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," in Computer Graphics Forum, 2021, vol. 40, no. 2, pp. 683–712.
[7] J. Palacios and J. Triska, "A comparison of modern gpu and cpu architectures: And the common convergence of both," Oregon State University, 2011.
[8] S. G. Parker et al., "Optix: a general purpose ray tracing engine," Acm transactions on graphics (tog), vol. 29, no. 4, pp. 1–13, 2010.