RasterPIP: Answering Point-in-polygon Query with GPU-native Transformation and Rasterization

Ziang Liu[1], Hui Li[2,3], Yingfan Liu[2]✉, Hua Tong[1], Zhenning Shi[2], Hui Zhang[3], and Jiangtao Cui[2]

[1] School of Cyber Engineering, Xidian University, Xi'an, China
[2] School of Computer Science and Technology, Xidian University, Xi'an, China
{zaliu1,htong,znshi}@stu.xidian.edu.cn
{hli,liuyingfan,cuijt}@xidian.edu.cn
[3] Shanghai Yunxi Technology Company Limited, Shanghai, China
zhanghui@inspur.com

Abstract. With advancements in location sensing, spatial data is increasing in volume and value. A fundamental problem in spatial databases, the point-in-polygon (PIP) query, returns points within a (set of) given polygon(s) and has broad applications in location-based services. Using the GPU rendering pipeline to replace traditional PIP tests is a promising approach. However, due to the limited color channel space per pixel, this paradigm faces challenges in effectively distinguishing the IDs of points and polygons. We address these challenges by categorizing PIP queries into two types. Based on the characteristics of each type, we study the impact of different rendering methods when converting PIP tests into object-rendering operations. This enables us to judiciously choose the storage modes for point and polygon IDs, tailored for each type. Extensive experiments show that our method effectively leverages the GPU's native rendering capabilities, outperforming existing solutions by one to three orders of magnitude.

Keywords: Spatial databases · GPU-native operations · Spatial query.

1 Introduction

With spatial databases and GIS systems, we can efficiently manage, query, and analyze spatial data to derive valuable insights, thereby enabling various location-based services [2–4]. For instance, geofencing [16], as a key technology supporting proactive location-based services, requires setting one or more virtual areas with defined boundaries to actively detect users entering or leaving these areas, so that notifications and messages can be pushed toward. Accordingly, the Point-in-Polygon (PIP) query, determining whether the system can accurately locate users in real time, is a fundamental task in geofencing. For example, geofencing can be applied in multi-zone alerting scenarios. The platform may detect all users (may not need to tell which zone a user belongs to)

within zones of danger buildings and push alert towards them when there is an earthquake. As another example, the platform can also serve to detect infectors within a series of campus zones, keeping the propagation of virus under control in real-time.

To be formal, each user could be denoted as a spatial point that represents its current location, while each zone can be viewed as a spatial polygon. Let D be the set of spatial points and P be the set of spatial polygons; thus finding users located in specific target zones is, in fact, asking for 1) the set of points in D falling in at least one polygon in P; 2) the complete mapping between $o \in D$ and $p \in P$ such that o falls within p. Note that the PIP problem here deals with multiple points and polygons simultaneously. In order to distinguish the two scenarios listed above, we call them boolean PIP (bPIP) and integral PIP (iPIP), respectively. To make it clear, we illustrate both in the form of SQL queries as follows (Query 1 and 2). Obviously, both are spatial selection queries that involve a spatial join between D and P, but with a different projection list.

```
Query 1 (example of bPIP):
SELECT DISTINCT D.id
FROM D, P
WHERE P.region CONTAINS D.location;

Query 2 (example of iPIP):
SELECT D.id, P.id
FROM D, P
WHERE P.region CONTAINS D.location
ORDER BY P.id;
```

```
Query 3:
SELECT count(*)
FROM D, P
WHERE P.region CONTAINS
  D.location
GROUP BY P.id;
```

Among existing PIP solutions, RayCasting [10] is the most popular. The distinct geometric properties exhibited by points inside and outside a polygon are used to test if a point is inside a polygon, which is known as a PIP test. Therefore, we can decompose both bPIP and iPIP into a series of PIP tests via RayCasting. However, as it is sensitive to the number of points and polygons, as well as the sides number of each polygon, RayCasting does not scale well if these factors increase. To address that, PostGIS[1], a popular open-source geographic engine on the basis of PostgreSQL, employs R-tree [11], combined with the Minimum Bounding Rectangle (MBR) of query polygons, to pre-filter points that are definitely outside the polygons, thereby reducing the number of PIP tests. However, when the data volume or polygon areas are large, the candidate points after filtering may still be numerous. In this case, a large number of PIP tests is still inevitable, leading to poor performance [15].

Recently, GPU-aware approaches [5, 6, 14, 22] have attracted research attention. RasterJoin [22] is the pioneering work in answering spatial *aggregation* queries as shown in **Query 3**. RasterJoin leverages GPU-native rasterization to replace PIP tests by rendering points and polygons on the same canvas and testing if they overlap on the same pixel. This approach applies to spatial aggregation queries, without the need for materializing intermediate spatial join results. RasterJoin utilizes the color channel of each pixel to store aggregated information. However, this cannot directly answer spatial selection queries such

[1] https://postgis.net/

as **Query 1** and **Query 2**. For aggregation queries, the information stored in each pixel does not need to be *distinguishable*, such as COUNT and SUM. In comparison, **Query 1** and **Query 2** need to explicitly output the IDs of the points or polygons. Therefore, it is necessary to ensure that the values in the color channel can distinguish these IDs. A simple way is to represent IDs by one-hot encoding. However, given the limited space of color channels and possibly large IDs in real applications, this method is impractical. [5, 6] showcase the advantages of GPU in answering spatial queries. However, it does not provide an efficient solution for distinguishing multiple data points on the same pixel. GPU-Reg [14] transforms spatial selection queries into spatial aggregation ones by replication and encoding. However, it is costly due to the iterative replication.

The above suggests that GPU is promising to efficiently answer PIP queries. The key challenge here is how to make full use of the limited built-in variables in GPU to efficiently store and distinguish crucial information, *i.e.,* the IDs. To overcome the above challenges, inspired by RasterJoin and GPU-Reg, we introduce RasterPIP, a novel approach using GPU-native transformation and rasterization to solve both bPIP and iPIP. Following the paradigm in RasterJoin, we transform the PIP tests into a series of object-rendering operations on the GPU. For various PIP problems, we study the impact of different rendering methods and judiciously choose the storage modes for the IDs of points and polygons within the GPU. This allows us to distinguish those IDs while minimizing unnecessary resource consumption. Extensive experiments show that we successfully leverage the GPU's native rendering capabilities to answer PIP queries, achieving $1 \sim 3$ orders of magnitude performance improvement.

Our technical contributions in this work can be summarized as follows.

- We propose a novel solution, RasterPIP-b to solve bPIP problem on the basis of GPU-native rendering pipeline.
- We propose a novel solution, RasterPIP-i to solve iPIP problem based on GPU-native rendering pipeline. We theoretical prove that the number of pixel rendered is much less than the state-of-the-art GPU-based solution.
- We conducted extensive experiments on real-world datasets to demonstrate the advantages of our methods.

The rest of this work is organized as follows. In Section 2, we introduce the preliminaries required for our solution. Afterwards, we present the details of RasterPIP-b and RasterPIP-i in Section 3 and Section 4, respectively. Experimental results are reported in Section 5. We review related works in Section 6. Finally, in Section 7 we conclude this paper and outline future work.

2 Preliminaries

In this section, we first formalize the two PIP problems, and then introduce the native GPU operations exploited in our solution.

2.1 Problem definition

Definition 1 ***Boolean Point-in-Polygon (bPIP) Query.*** *Given a set $D \subset \mathbb{R}^2$ and a set P of polygons in $\mathbb{R}^2$, bPIP query returns a set $R \subset D$, such that $\forall o \in R$, $\exists p \in P$ and p contains o.*

That is, we need to tell whether or not an arbitrary point o is contained by P and hence it is referred to as *boolean* PIP problem. This problem can be easily found in various applications [14]. Differently, in some other applications [9], we may also wonder *which* polygon o is contained by, as shown below.

Definition 2 ***Integral Point-in-Polygon (iPIP) Query.*** *Given a set $D \subset \mathbb{R}^2$ and a set P of polygons in $\mathbb{R}^2$, iPIP query returns a set $S = \{(o, p) | o \in D, p \in P$ and p contains $o.\}$*

Compared with bPIP, iPIP returns the IDs (*i.e.,* integers) of polygons each point o is contained by. Therefore, it is referred to as Integral PIP. In the above definition, since PIP queries are typically performed in the plane, the algorithms discussed in this paper are primarily applicable to two-dimensional spaces.

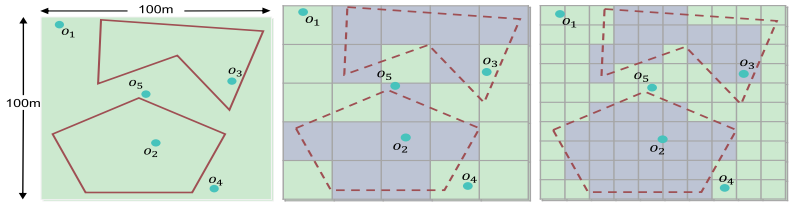
2.2 GPU-native operations

In this part, we introduce two GPU-native operations we adopt in this work.

Polygon Drawing. Drawing a polygon by the GPU is called polygon rendering, involving decomposing a polygon into a set of triangles and rendering them on a screen with a specific resolution through rasterization. In this paper, we employ the constrained Delaunay polygon triangulation [18], but alternative methods could also be used as this is not the focus of our work.

As a cross-language and cross-platform API for rendering graphics, OpenGL [13] provides two types of programmable shader, the *i.e.,* vertex shader and the fragment shader to render a triangle. The former conducts the transformation of each vertex to the screen space, while the latter processes the attributes of fragments covered by the triangle, such as fragment colors.

Point Drawing. As another key operation concerned in this work, the process of drawing a point is similar to drawing a vertex of a triangle, as discussed above. According to OpenGL [13], we can follow the vertex shader program to implement this. To answer bPIP and iPIP, we do not need to physically display the rendering result on the screen, but we need OpenGL modules to store the immediate results *w.r.t.* each pixel during rendering. Luckily, we find **framebuffer object (FBO)** and **shader storage buffer object (SSBO)** in OpenGL for this purpose. In general, FBO is the object that creates and manages framebuffers for off-screen rendering. In our context, a framebuffer can be viewed as a pixel matrix. Given a specified resolution such as $2560px \times 1440px$, the corresponding framebuffer contains 2560×1440 pixels, each of which is represented by four 32-bit values *w.r.t.* 4 color channels (red, green, blue and alpha). In this work, we would like to use the 128-bit storage for each pixel to record important information such as IDs of points and polygons. Unlike framebuffer, which stores data for each pixel, SSBO could be used to store data with a large size (up to the available GPU memory). In this work, the results are temporarily stored in the GPU memory and finally returned to the main memory. SSBO is more suitable to our needs due to its large capacity.



(a) Spatial objects (b) $\epsilon = 20$ meters (c) $\epsilon = 10$ meters

Fig. 1. Illustration of the effects of ϵ on Rasterization.

3 RasterPIP-b

3.1 Basic Strategy

As defined in Def. 1, a bPIP query returns a set of points that are inside at least one polygon. The traditional two-phase approach, *i.e.,* filtering and refining, becomes costly when there are many points and polygons. The excellent parallel processing capabilities of GPU offer an opportunity to address bPIP queries. Considering the native rendering operations of GPU introduced in the last section, we can address the problem by mapping the points and polygons to the same canvas. Therefore, the PIP test can be transformed into a test of intersection between points and polygons on that canvas, which can be seen as a matrix of pixels. By checking whether a point and a polygon cover the same pixel, one can determine whether the point is inside that polygon. This paradigm was used to effectively solve spatial aggregation queries in [22]. However, as we introduced in Section 1, it cannot directly answer the bPIP query. Unlike spatial aggregation queries, bPIP queries need to return the point IDs. Since points can be numerous and their IDs can be large, the key challenge is *how to effectively distinguish IDs on the same pixel with limited storage space*. Fortunately, compared to point IDs, under the bPIP problem, we do not need to differentiate between polygons. Moreover, an important insight is that during a canvas test, different rendering orders can all capture the spatial relationship between points and polygons. However, only spatial objects rendered first need to store information in the pixel, so that later rendered objects can determine whether they cover the same pixel based on this information. Inspired by these observations, we propose RasterPIP-b, which utilizes GPU-native transformation and rasterization operations to solve the bPIP problem.

Before diving into the details, we need to introduce the concept of *spatial resolution*, *i.e.,* the smallest unit that can be clearly identified in an image [23], that is, the actual distance of the ground represented by each pixel in meters, a.k.a. *pixel size* in some work [12]. We denote the pixel size as ϵ. The smaller ϵ, the higher the required resolution.

Example 1. Consider the real-world spatial objects in Fig. 1(a), which contains two polygons and five points in a $100m \times 100m$ rectangular region. We illustrate the effects of ϵ in Fig. 1(b) and Fig. 1(c), where ϵ is set to 20 meters (ground distance between the center of two adjacent pixels is 20 meters) and 10 meters, respectively. Obviously, a 5×5 pixel matrix is enough for Fig. 1(b), while 10×10

Algorithm 1: RasterPIP-b

```
   Input: the set of points and polygons, D and P; the pixel size ε
   Output: R
 1 // Procedure of DrawPolygons^b
 2 set the resolution of the FBO F_pg based on ε;
 3 initialize the FBO F_pg as 0;
 4 for each p ∈ P do
 5 |   triangulate p to obtain a set T of triangles;
 6 |   for each t ∈ T do
 7 |   |   convert the vertices of t to the screen space;
 8 |   |   rasterize t into a set F of fragments;
 9 |   |   for each (x, y) ∈ F do
10 |   |   |   F_pg(x, y) = 1;
11 |   |   end
12 |   end
13 end
14 // Procedure of DrawPoints^b
15 R = ∅;
16 for each o ∈ D do
17 |   transform o to get its position (x, y) in the screen space;
18 |   if F_pg(x, y) == 1 then
19 |   |   R = R ∪ {o};
20 |   end
21 end
22 return R
```

for Fig. 1(c). As a result, the resolution of the canvas is $5px \times 5px$ and $10px \times 10px$, respectively. Higher resolution leads to more accurate rendering results.

Since we do not require the IDs of the polygons, given a specific ϵ, our method first draws all polygons in P on a canvas by transforming each polygon from the original space to the screen space, and we only need to record whether a pixel is covered by polygons by simply storing a '1' in the color channel. Subsequently, we draw each point $o \in D$ on the same canvas. Recognizing the pixel hit by o, we can identify whether or not the same pixel has been covered by any polygon, by checking if its corresponding color channel value is 1. If so, o is identified as a valid result, whose ID can be directly stored in SSBO and finally returned to the main memory, without writing it to the color channel. The pseudocode of our method is shown in Algorithm 1. Due to space constraints, some algorithms, proofs, and examples are not detailed in this paper, but are thoroughly explained in the corresponding sections of our technical report [1].

Example 2. We present an example illustrating the pipeline of Algorithm 1 in Fig. 2. Assuming that ϵ is specified to 20 meters, the required resolution is $5px \times 5px$. We first draw the polygons. For each pixel covered by the polygons, write '1' to its color channel, *e.g.,* $F_{pg}(5, 5)$. Next, each point is transformed to the screen space and the corresponding coordinates are obtained. Based on the coordinates, we can check the value of the corresponding pixel color channel. For example, for o_1, $F_{pg}(1, 5) = 0$, indicating that it is not covered by the polygons; but for o_2, the color channel value is 1. Therefore, its ID is recorded in SSBO. Through the above process, o_2 and o_5 are ultimately identified.

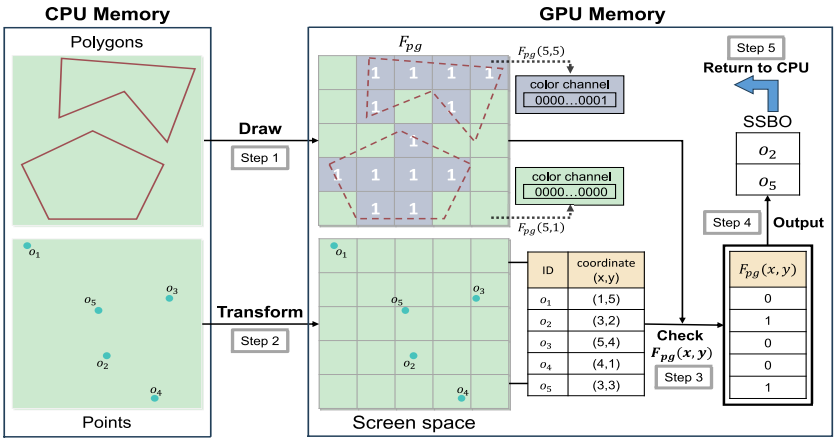


Fig. 2. Illustrating the pipeline of RasterPIP-b.

Interestingly, although the solution is efficient, there may be false positives and false negatives in the set of results R. In example 2, o_3 is a false negative while o_5 is a false positive, which can be easily observed in Fig. 1(b). This is due to hardware limitations during the rasterization process. In general, a higher resolution reduces the likelihood of such errors, leading to more accurate results. As shown in Fig. 1(c), compared to $\epsilon = 20$, RasterPIP-b with $\epsilon = 10$ correctly classifies both o_3 and o_5 and thus avoids returning false positives and negatives.

However, a higher resolution requires more pixels to draw the polygons. In fact, the number of pixels covered by polygons and points is key to the cost of RasterPIP-b, since it has to deal with each of these pixels. The time complexity is provided in Theorem 1. Therefore, ϵ is the key parameter to balance effectiveness and efficiency within our solutions, which should be carefully selected.

Theorem 1. *Given a set of polygons P and a set of points D, let m and n be the number of polygons and the number of points, respectively. Each polygon in P has an average of v vertices and the average area A of its minimum bounding rectangle (MBR). Suppose ϵ refers to the pixel size; then the upper bound of the time complexity for RasterPIP-b is $O(m \cdot (v \log v + A/\epsilon^2) + n)$.*

3.2 Partitioned RasterPIP-b

In some scenarios, users may have a high demand for the accuracy of the results. As mentioned in the previous subsection, we can increase the resolution by reducing the value of ϵ, thus improving the accuracy. However, when ϵ becomes sufficiently small, rendering a large area requires a considerable number of pixels, which may exceed the maximum resolution supported by the GPU. For example, suppose that we are performing bPIP over the rectangular region in Fig. 1(a) and the maximum resolution supported by the GPU is $50px \times 50px$. When ϵ is set to 1 meter, the number of pixels needed can be as large as $100px \times 100px$. In this case, the canvas cannot cover all points and polygons. To address this, we partition the points and polygons based on the required and maximum resolution, determining the minimum number of canvases needed to fully render the region. We then split the region evenly and render each part separately, allowing

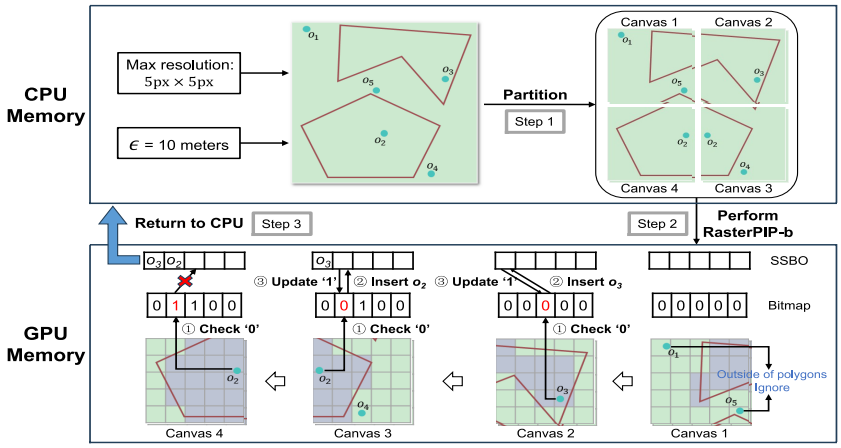


Fig. 3. Illustrating the pipeline of partitioned RasterPIP-b.

each part to be drawn with the required resolution. RasterPIP-b is applied to each canvas individually, and finally we merge the results from all partitions.

However, the results obtained from the partition strategy sometimes contain duplicate points. Specifically, points located precisely on the partition boundaries will be included in adjacent parts at the same time, leading to duplicate results. A naive way to solve this problem is to remove duplicates by traversing the result set. However, as the number of points increases, it will produce additional overhead. Therefore, we need to avoid redundancy in the result set as early as possible. Specifically, we employ a bitmap for all points at run-time to indicate whether a point has already been identified as inside the polygons. If it has, there is no need for further evaluation, thereby avoiding duplicates in the result set. An example of the whole partition procedure is illustrated in Fig. 3, where o_2 is the point located at the partition boundaries.

4 RasterPIP-i

In this section, we present a novel solution towards iPIP. Intuitively, the iPIP problem can be decomposed into a series of bPIP problems and solved by invoking RasterPIP-b multiple times. However, this approach is inefficient, so we propose RasterPIP-i tailored to the iPIP problem.

4.1 Details of RasterPIP-i

The iPIP query in Def. 2 not only finds out the points that fall inside at least one polygon, but also specifies which polygon(s) each point falls inside. Therefore, both the IDs of points and polygons are required, and thus at least one type of ID should be recorded in the color channel for later use. However, this is blocked by the limited representation capability in a pixel's color channel. Thus, we propose RasterPIP-i, which adopts a batch processing approach to overcome the above challenge.

Like RasterPIP-b, RasterPIP-i needs to draw the polygons and the points, but with a slight difference, denoted as $\texttt{DrawPolygons}^i$ and $\texttt{DrawPoints}^i$,

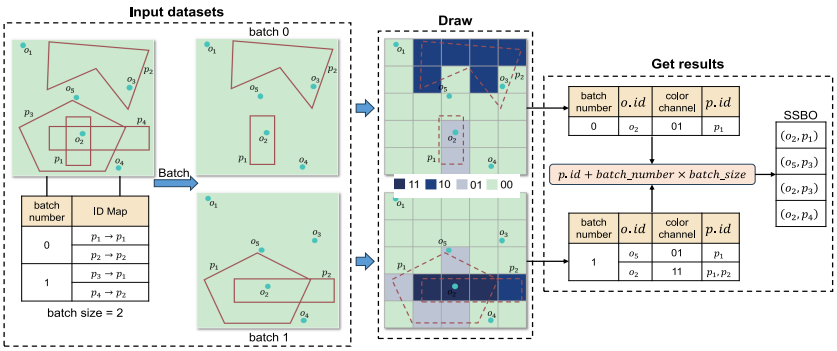


Fig. 4. Illustrating the pipeline of RasterPIP-i.

respectively. In $\texttt{DrawPolygons}^i$, we need to draw each polygon in P to get the set of pixels covered by it. In addition, for a pixel with coordinate (x, y), we should store the IDs of the polygons that cover it. First, let us assume $|P| \leq 128$ (we shall remove this assumption soon) and that the polygon IDs are in the range $[0, |P|)$. We use the FBO F_{pg} to store the corresponding polygon IDs for each pixel, which corresponds to 4 32-bit color channels. For each pixel, we could use this 128-bit bitmap to represent any subset of P covering it. If a polygon with ID i covers a pixel, we set the i-th bit of its 128-bit bitmap as 1. During $\texttt{DrawPolygons}^i$, we update F_{pg} for each pixel. Like $\texttt{DrawPoints}^b$, $\texttt{DrawPoints}^i$ first transforms each point $o \in D$ into the screen space and gets the pixel (x, y) hit by o. Then, we can recognize the subset of polygons that cover the pixel by decoding the 128-bit bitmap *w.r.t.* $F_{pg}(x, y)$.

Now, let us consider $|P| > 128$. Since the 128-bit bitmap for each pixel can represent at most 128 polygons, we divide P into batches, where each batch contains at most 128 polygons. For a batch of 128 polygons, we encode each polygon ID id in a new range between 0 and 127 by $id\%128$, so that the 128-bit bitmap could be used to solve iPIP as discussed above. Obviously, we can easily decode an ID in the new range from its original ID. Then, we obtain the point-polygon pair by first drawing the polygons in the batch and then drawing each point. The final results are merged once all batches have been processed in the same way. To be formal, we provide the pseudocode for RasterPIP-i in Algorithm 2, in which the EncodeID and DecodeID function are responsible for ID mapping as mentioned earlier. Moreover, there is an example workflow in Fig. 4, assuming the color channel contains only two bits. Therefore, the four given polygons need to be processed in two batches.

4.2 Theoretical comparison of RasterPIP-i and GPU-Reg

In Section 4.1, RasterPIP-i renders polygons first. An alternative approach is to render points first, which can yield the same results, as demonstrated in GPU-Reg. However, different rendering orders result in different batches of geometric objects, leading to different total numbers of rendering pixels, which is the key difference between them. Let us denote the number of polygons and points as m, n, respectively, A as the average area of the MBR for each polygon, B as the batch size, and N as the number of pixels rendered. Obviously, all polygons

Algorithm 2: RasterPIP-i

```
   Input: the set D of points and the set P of polygons
   Output: S
 1 B = ⌈|P|/128⌉;
 2 for j = 0 to B − 1 do
 3 |   // Procedure of DrawPolygons^i
 4 |   initialize F_pg as 0;
 5 |   for each p ∈ P_j do
 6 |   |   newPolyID = EncodeID(p.id);
 7 |   |   obtain the set FS of fragments hit by p after triangulation and rasterization;
 8 |   |   for each (x, y) ∈ FS do
 9 |   |   |   F_pg(x, y) = F_pg(x, y) OR (1 << newPolyID);
10 |   |   end
11 |   end
12 |   // Procedure of DrawPoints^i
13 |   S = ∅;
14 |   for each o ∈ D do
15 |   |   transform o to get its pixel position (x, y);
16 |   |   originalPolyids = DecodeID(F_pg(x, y), j);
17 |   |   for each p.id ∈ originalPolyids do
18 |   |   |   S = S ∪ (o.id, p.id);
19 |   |   end
20 |   end
21 end
22 return S
```

involve $m \cdot A/\epsilon^2$ pixels at most. Considering that rendering each pixel has the same cost, we compare both methods in the total number of pixels rendered.

RasterPIP-i batches the polygons, resulting in repeatedly rendering of all points in each batch, while each individual polygon is rendered only once. Therefore, the number of pixels rendered in RasterPIP-i is $N_1 = A \cdot m/\epsilon^2 + \lceil m/B \rceil \cdot n$. In contrast, GPU-Reg batches points, causing repeated rendering of all polygons in each batch, while all points are rendered once. Therefore, the number of pixels rendered in GPU-Reg is $N_2 = n + \lceil n/B \rceil \cdot A \cdot m/\epsilon^2$. As a result, $N_1 - N_2 = (\lceil \frac{m}{B} \rceil - 1) \cdot n - (\lceil \frac{n}{B} \rceil - 1) \cdot \frac{A \cdot m}{\epsilon^2}$. Obviously, in practice, $m \ll n$. Therefore, GPU-Reg requires much more batches than RasterPIP-i. In addition, it is reasonable to assume that the number of pixels covered by all polygons is approximately equal to or even larger than n, especially considering that the points have been filtered by a spatial index. Therefore, in practice, $N_1 \ll N_2$, which demonstrates the advantages of our method over GPU-Reg in the number of pixels rendered. Hence, our method, theoretically, has a significant advantage over GPU-Reg in efficiency.

5 Experiments

5.1 Experimental settings

Datasets. We conduct experiments on two real-word datasets, *OSM* and *Taxi*. *OSM* is the Open Street Map dataset provided by [8]. We randomly sampled 10 million points from *OSM* within the United States. Each point is associated with a unique ID and geographical coordinates. *Taxi*[2] records the trip data of yellow taxis in New York City, including pick-up and drop-off locations, travel

[2] https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

distances, etc. We select the trip data for one month and extract the pickup locations to form the set of points D. There are 13,294,789 points in *Taxi*. Both the above datasets are skewed. For instance, the points in *Taxi* are extremely crowded around Manhattan. In light of that, to study the impact of varied data distributions, we extraly generated two datasets, each containing 10,000,000 points within the scope of New York City, with normal and uniform distribution, respectively. We shall refer to them as *distribution* dataset.

In terms of polygons, we utilize both real-world regions from *Taxi Zones* and *Synthetic polygons*. *Taxi Zones*[3] represent the division of New York City taxi regions, consisting of 263 polygons, most of which are concave in shape, and each polygon represents a taxi service zone. *Synthetic polygons* are provided by RasterJoin [22], with the aim of generating a substantial number of polygons with properties similar to the real ones, and contain many simple and complex polygons with various shapes (concave and convex) and number of sides.

Environments. All experiments are carried out on a server with two Intel Xeon E5-2680 CPUs, 256 GB of RAM and a Nvidia 2080TI GPU with 11 GB of memory. The solutions are implemented using C++ and OpenGL.

Performance Measures. Each method is evaluated from both the efficiency and the precision aspect. The former is measured by the elapsed time, denoted as *cost*, and the latter is evaluated by the F_1 score [20].

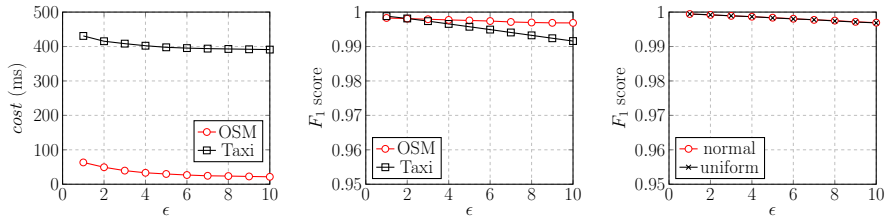
Compared Baselines. We compare our methods with CPU-based methods, including **RayCasting**, **PostGIS**, as well as the latest GPU-based solution, *i.e.,* **GPU-Reg** [14]. **RayCasting** is a popular CPU-based solution [10] and is paralleled with *OpenMP*[4]. **PostGIS** provides ST_Within function to identify points inside polygons, which uses R-tree to filter irrelevant points. We set the maximum parallel workers of PostGIS as the number of CPU threads. **GPU-Reg** is the latest solution based on the rendering pipeline of GPUs.

For both RasterPIP-b and RasterPIP-i, we limit GPU memory to 4GB, set the maximum FBO size to $8192px \times 8192px$, and vary ϵ between 1 and 10 meters. For RasterPIP-i, we set the batch size to 128. For GPU-Reg, we set its ϵ to 10 meters. Both RasterPIP and GPU-Reg use a simple MBR for coarse-grained point filtering and the triangulation time has already been taken into account. In particular, all experimental results are averaged over ten runs.

5.2 Experimental results for RasterPIP-b

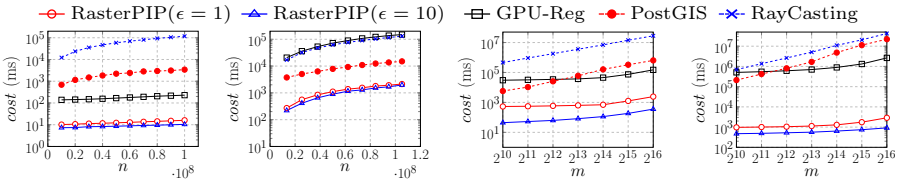
Effects of the Pixel Size (ϵ). Due to the hardware limitation of GPU, RasterPIP-b may produce some incorrect results, mainly caused by ϵ, as discussed in Section 3. To investigate the effects of ϵ on RasterPIP-b, we randomly select 5 to 50 polygons from *Taxi Zones* to form the set P, and vary ϵ between 1 and 10 meters. Fig. 5(a) and Fig. 5(b) show the results. In general, both *cost* and F_1 decrease as ϵ increases. This is because a smaller pixel size results in more pixels to be rendered, leading to higher cost but more precise results. In addition, the F_1 scores are always larger than **0.99**, which means that the error introduced

[3] https://data.cityofnewyork.us/Transportation/NYC-Taxi-Zones/d3c5-ddgc
[4] https://www.openmp.org/



(a) *cost* vs ϵ (b) F_1 score vs ϵ (c) various distribution

Fig. 5. The effects of the pixel size ϵ on RasterPIP-b.



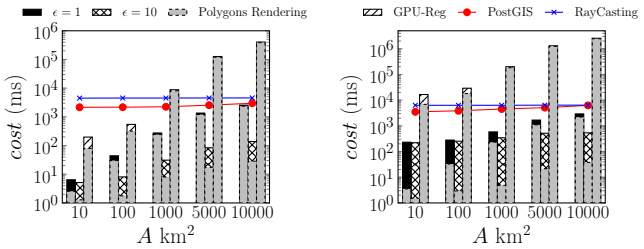
(a) OSM (b) Taxi (c) OSM (d) Taxi

Fig. 6. (a, b) The effects of n; (c, d) the effect of m on RasterPIP-b.

by the hardware is extremely limited in our solution. To be more convincing, the experimental results of the *distribution* dataset in Fig. 5(c) show that the F_1 for the two distributions are almost indistinguishable and still remain above 0.99 even when ϵ is set to 10. Note that we did not separately evaluate the performance of the partition-based RasterPIP-b, as it serves to extend the current resolution to meet pixel size requirements. Thus, the effectiveness of partition is implicitly reflected in the experiments on pixel size: when the pixel size is small, the partition is necessarily to expand the resolution, thereby further limiting the errors of hardware resolution. In the rest of this part, for ease of discussion, we report the results under two pixel sizes, *i.e.*, $\epsilon = 1$ and $\epsilon = 10$, respectively.

Effects of the Number of Points (n). To test the scalability of RasterPIP-b *w.r.t.* n, we vary the number of points sampled from *OSM* from 10 million to 100 million. Furthermore, we randomly select five polygons from *Taxi Zones* to form the set P. The results are reported in Fig. 6(a-b). RasterPIP-b consistently outperforms other approaches by at least one order of magnitude. The superiority of RasterPIP-b in *cost* is mainly due to the fact that rendering a single pixel by the GPU-native operations is significantly lower than that of a PIP test used by CPU-based methods. On the other hand, GPU-Reg needs to render the same query polygons repeatedly over multiple batches, which is linear to n. As a result, it has to re-render more pixels as n increases. In comparison, RasterPIP-b only renders the polygons *once* during the whole process. Note that GPU-Reg has good performance on *OSM*, but much worse on *Taxi*. This is because the filtering efficiency of the MBR on *OSM* is much better than that on *Taxi*, thereby reducing the frequency with which polygons are rendered repeatedly.

Effects of the Number of Polygons (m). We utilize *Synthetic polygons* to form P such that $m \in [2^{10}, 2^{16}]$. Fig. 6(c-d) shows *cost* by varying m. RasterPIP-b outperforms other methods with both $\epsilon = 1$ and $\epsilon = 10$. The cost of RasterPIP-



(a) OSM (b) Taxi

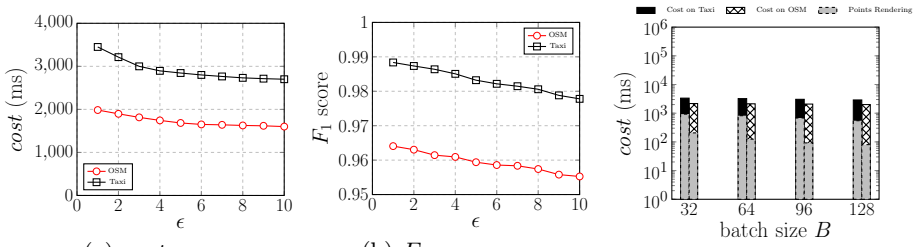
Fig. 7. The effects of the area A on RasterPIP-b.

b is closely related to the number of rendering pixels, as discussed in Section 3.1. Given the pixel size ϵ, the larger area P covers, the more pixels are rendered and the more time RasterPIP-b spends. However, as shown in Fig. 6(c-d), the more polygons there are, the more expensive PIP tests must be performed, resulting in a rapid increase in the *cost* of CPU-based methods. For GPU-Reg, due to the repeated rendering of polygons, as the number of polygons increases, the growth in *cost* is higher than that of RasterPIP-b. Overall, RasterPIP-b is less sensitive to the number of polygons compared to other methods.

Effects of the Area of Polygons (A). As discussed in Section 3.1, given ϵ, the average area A of the polygons directly affects the number of pixels rendered by RasterPIP-b and further impacts *cost*. To justify that, we select a rectangle to form P and vary its area between 10 and 10000 km^2 on both *OSM* and *Taxi*. The results are shown in Fig. 7. The pure gray portion of each bar represents the cost of polygon rendering in GPU-based methods. Notably, RasterPIP-b only needs to render the polygon once, thus incurring a lower cost compared to GPU-Reg. Furthermore, as the area increases, the increase in cost is not significant. In addition, RasterPIP-b has a low rendering cost due to the large size of pixels $\epsilon = 10$, since only a few pixels are processed.

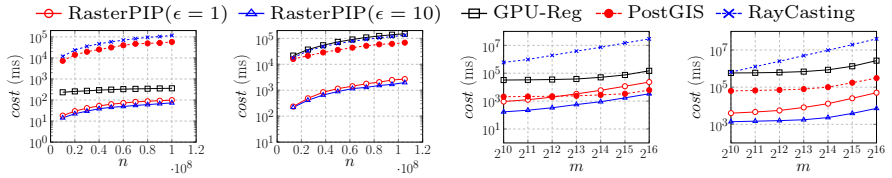
In comparison, RayCasting conducts PIP tests for each point, whose computational cost is determined by the number of sides of the polygon and is not sensitive to A. PostGIS employs an index to obtain candidates and then performs PIP tests for each candidate. Since the area of the polygon has a positive impact on the number of candidates, PostGIS spends more time on a larger polygon. In Fig. 7, the cost of RasterPIP-b remains consistently the lowest.

It is worth noting that on *OSM*, as the area increases, the cost of PostGIS and RasterPIP-b becomes increasingly closer when ϵ is 1. This is due to the broader coverage of the *OSM*. As the area grows, the increase in the number of candidate points is not significant, leading to only a modest increase in the number of PIP tests for PostGIS. This reveals a potential issue with RasterPIP-b: it might not perform well for a polygon that has a large area but contains few points. We simulate an extreme scenario where the rectangular area is 10000 km^2 but covers only 20 points. Specifically, when ϵ is 1, the cost of RasterPIP-b is 4038 ms, while PostGIS takes only about 1.5 ms. Although such special situation is rare in the real-world application such as geofencing, it still indicates that RasterPIP-b may not always be superior. Therefore, it is necessary to design a cost model to decide



(a) *cost* vs ϵ (b) F_1 score vs ϵ (c) batch size B

Fig. 8. The effects of pixel size ϵ (a,b) and batch size B (c) on RasterPIP-i.



(a) OSM (b) Taxi (c) OSM (d) Taxi

Fig. 9. (a, b): effect of n; (c, d) effect of m on RasterPIP-i.

whether to use RasterPIP-b based on specific scenarios. Fortunately, when ϵ is 10, the cost of RasterPIP-b is acceptable, *i.e.,* 22 ms, allowing for quick query responses when precise results are not required.

5.3 Experimental results for RasterPIP-i

Effects of Pixel Size (ϵ). Like RasterPIP-b, as ϵ increases, both *cost* and F_1 score decrease, but the F_1 score remains above 0.95 in Fig. 8(a-b). Next, we will evaluate RasterPIP-i at two representative pixel sizes, *i.e.,* 1 and 10.

Effects of Batch Size (B). Since each pixel has four 32-bit color channels, we vary the batch size of polygons in $\{32, 64, 96, 128\}$. In Fig. 8(c), as B increases, *cost* for rendering points decreases, since a larger batch size requires fewer iterations, where all the points are rendered per iteration. In the following, we set B to 128 by default.

Effects of n and m. We employ the same settings as the corresponding experiments in Section 5.2. The results in Fig. 9 show that RasterPIP-i outperforms other methods in almost all cases. Note that, on *OSM*, as m increases, the performance of PostGIS is comparable to that of RasterPIP-i with $\epsilon = 1$. The reason is, for *OSM*, few points remain after index filtering, leading to few subsequent PIP tests. However, our method is significantly superior when $\epsilon = 10$.

5.4 Extensive experiments

We also conduct experiments to investigate the sensitivity of RasterPIP-b (*resp.,* RasterPIP-i) to the side number of polygons, as well as the shape, *i.e.,* convex or concave. In general, RasterPIP-b (*resp.,* RasterPIP-i) significantly outperforms all other baselines and exhibits three interesting properties. First, it is able to balance efficiency and accuracy by adjusting the pixel size ϵ. Second, it is low sensitive to either the number of points or the number of polygons. Third, it is low sensitive to the shape, area or side number of query polygon. Details about the above results can be found in [1].

6 Related works

CPU-based solutions for PIP. The representative attempts for PIP, like *RayCasting* and *AngleSummation*, can be found in computational geometry [10,19]. These methods face challenges with large spatial data, leading to the development of spatial databases [17], into which PIP problem solutions have been incorporated. These solutions involve two steps: filtering and refinement. In the filtering step, spatial indices, *e.g.,* R-trees [11], are adopted to filter out those points that are definitely not inside the query polygon. In the refinement stage, only the remaining candidate points need to be involved in PIP tests, thereby reducing the overhead. Other CPU-based methods, such as IDEAL [21], utilize the characteristics of the raster model to guide spatial queries over the vector model. Rasterization Filter [26] proposes a raster approximation for processing spatial joins.

GPU-based solutions for PIP. Due to the parallel computing capability of GPU, there exist a series of efforts [7] that utilize the GPU to accelerate PIP queries. [25] utilized GPU to design an indexing method based on grid files in the filtering phase and a parallel PIP testing scheme in the refinement phase; [24] proposed a pair of methods for constructing quadtrees based on GPU.

Notably, the aforementioned algorithms utilize the parallel computing scheme of GPU, but still follow the design ideas and techniques of CPU methods. Recent studies [5, 6, 22] have shown that these methods fail to fully exploit the capabilities of the GPU and are complex to implement; algorithms relying on GPU-native rendering operations can make better use of GPU. RasterJoin [22] converts the point-in-polygon join operation into a rendering operation, answering spatial aggregation queries within arbitrary polygon areas. [5] conceptually designed a geometric data model and a series of algebras for spatial databases on GPU and demonstrated its expressiveness through a conceptual prototype. Our method might be an option for one of its operators in the future.

7 Conclusion

In this paper, we study two variants of point-in-polygon queries, *i.e.,* boolean point-in-polygon (bPIP) and integral point-in-polygon (iPIP). To efficiently answer both queries, we respectively present two novel solutions by leveraging on the native rendering capabilities of GPU. As justified in our empirical study on real-world datasets, compared to traditional CPU-based methods, our solutions are less sensitive to shape, side number, area, and total number of query polygons; compared to the GPU-based method, our method is significantly faster. Admittedly, the results of GPU-based solutions may not be exactly accurate, it is subjected to the hardware but not algorithmic design. Improvement of GPU hardware, which is beyond the scope of our work, definitely alleviate the problem.

Acknowledgments. This work was supported by Special Task Project of the Ministry of Industry and Information Technology of China (No. ZTZB-23-990-024).

References

1. Technical report, https://anonymous.4open.science/r/RasterPIP/

2. Chen, L., Gao, Y., Fang, Z.e.a.: Real-time distributed co-movement pattern detection on streaming trajectories. PVLDB **12**(10), 1208–1220 (2019)
3. Chen, L., Zhong, Q., Xiao, X., Gao, Y., Jin, P., Jensen, C.S.: Price-and-time-aware dynamic ridesharing. In: ICDE. pp. 1061–1072. IEEE (2018)
4. Ding, X., Chen, L., Gao, Y., Jensen, C.S., Bao, H.: Ultraman: A unified platform for big trajectory data management and analytics. PVLDB **11**(7), 787–799 (2018)
5. Doraiswamy, H., Freire, J.: A gpu-friendly geometric data model and algebra for spatial queries. In: SIGMOD. pp. 1875–1885 (2020)
6. Doraiswamy, H., Freire, J.: Spade: Gpu-powered spatial database engine for commodity hardware. In: ICDE. pp. 2669–2681. IEEE (2022)
7. Doraiswamy, H., Vo, H.T., Silva, C.T., Freire, J.: A gpu-based index to support interactive spatio-temporal queries over historical data. In: 2016 IEEE 32nd International conference on data engineering (ICDE). pp. 1086–1097. IEEE (2016)
8. Eldawy, A., Mokbel, M.F.: Spatialhadoop: A mapreduce framework for spatial data. In: ICDE. pp. 1352–1363. IEEE (2015)
9. Ferreira, N., Poco, J., Vo, H.T., Freire, J., Silva, C.T.: Visual exploration of big spatio-temporal urban data: A study of new york city taxi trips. IEEE transactions on visualization and computer graphics **19**(12), 2149–2158 (2013)
10. Galetzka, M.e.a.: A simple and correct even-odd algorithm for the point-in-polygon problem for complex polygons. arXiv preprint arXiv:1207.3502 (2012)
11. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD. pp. 47–57 (1984)
12. Hengl, T.: Finding the right pixel size. Computers & geosciences **32**(9), 1283–1298 (2006)
13. Kessenich, J., Sellers, G.e.a.: OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V. Addison-Wesley Professional (2016)
14. Li, H.e.a.: a2reginf: An interactive system for maximizing influence within arbitrary number of arbitrary shaped query regions. In: WSDM. pp. 1585–1588 (2022)
15. Nguyen, T.T.: Indexing postgis databases and spatial query performance evaluations. IJGI **5**(3), 1 (2009)
16. Reclus, F., Drouard, K.: Geofencing for fleet & freight management. In: 2009 9th ITST. pp. 353–356. IEEE (2009)
17. Samson, G.L., Lu, J., Usman, M.M., Xu, Q.: Spatial databases: An overview. Ontologies and big data considerations for effective intelligence pp. 111–149 (2017)
18. Shewchuk, J.R.: Delaunay refinement algorithms for triangular mesh generation. Computational geometry **22**(1-3), 21–74 (2002)
19. Sutherland, I.E., Sproull, R.F., Schumacker, R.A.: A characterization of ten hidden-surface algorithms. ACM Computing Surveys (CSUR) **6**(1), 1–55 (1974)
20. Taha, A.A., Hanbury, A.: Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. BMC medical imaging **15**, 1–28 (2015)
21. Teng, D., Baig, F.e.a.: Ideal: a vector-raster hybrid model for efficient spatial queries over complex polygons. In: MDM. pp. 99–108. IEEE (2021)
22. Tzirita Zacharatou, E., Doraiswamy, H.e.a.: Gpu rasterization for real-time spatial aggregation over arbitrary polygons. PVLDB **11**(3) (2017)
23. Zhang, J., Li, J.: Spatial Cognitive Engine Technology. Elsevier (2023)
24. Zhang, J., Gruenwald, L.: Efficient quadtree construction for indexing large-scale point data on gpus: Bottom-up vs. top-down. In: ADMS@VLDB. pp. 34–46 (2019)
25. Zhang, J., You, S.: Speeding up large-scale point-in-polygon test based spatial join on gpus. In: SIGSPATIAL. pp. 23–32 (2012)
26. Zimbrao, G., De Souza, J.M.: A raster approximation for processing of spatial joins. In: VLDB. vol. 98, pp. 24–27 (1998)