GPU Rasterization for Spatial Aggregate Join Queries

Eleni Tzirita Zacharatou[0000−0001−8873−5455]

1 Definition

A spatial aggregate join query is a fundamental operation in spatial data analysis used to combine spatially referenced attribute data (such as demographic statistics and environmental measurements) with spatial features represented as polygons (such as administrative boundaries and land parcels). This query applies a spatial predicate, such as containment or intersection, to associate the attribute data with the relevant spatial features. It then uses aggregate functions (e.g., sum, average, count) on the matched data, producing summarized results grouped by individual polygons.

In SQL notation, the query can be expressed as follows:

```
SELECT R.id, AGG(a_i)
FROM P, R
WHERE P.loc INSIDE R.geometry [AND filterCondition]*
GROUP BY R.id
```

Given a set of attribute data (points) of the form $P(loc, a_1, a_2, \dots)$, where loc and a_i are the location and attributes of the point, and a set of regions $R(id, geometry)$, the query performs an aggregation (`AGG`) over the result of the join between P and R. Functions commonly used for `AGG` include the count of points and average of the specified attribute a_i. A region's `geometry` can be any *arbitrary polygon*. The query can also have zero or more `filterConditions` on the attributes.

Traditionally, spatial joins are evaluated using a two-step "filter and refine" approach. The filter step employs simplified approximations, such as Minimum Bounding Rectangles (MBRs), to quickly eliminate object pairs that do not satisfy the spatial predicate. In the refinement step, more computationally intensive geometric tests are applied to the remaining pairs to verify the predicate using the actual geometries. For the given query, this refinement involves a Point-in-Polygon (PIP) test, which determines whether each data point lies within a polygonal region. The

Eleni Tzirita Zacharatou
Hasso Plattner Institute, Potsdam, Germany, e-mail: eleni.tziritazacharatou@hpi.de

computational cost of this test scales linearly with the number of polygon vertices. Consequently, it can become a significant bottleneck, as real-world polygons often contain hundreds of vertices. This challenge is further exacerbated by the sheer scale of modern datasets, which can include hundreds of millions to several billion points. As a result, traditional spatial join techniques, commonly used in database systems, are computationally expensive and are typically only suitable for batch processing [Tzirita Zacharatou et al.(2021)].

Evaluating spatial joins on GPUs presents unique challenges compared to traditional CPU-based methods, primarily due to GPUs' architectural differences and constraints. GPUs are designed for massively parallel processing, featuring thousands of cores capable of handling numerous operations simultaneously. In contrast, traditional CPU-based spatial joins often rely on serial or limited parallel processing. As a result, algorithms and data structures must be adapted to fully leverage GPU parallelism. Additionally, GPUs have limited on-board memory and require data to be transferred between the CPU and GPU memory, which can create a bottleneck. Efficient memory management and data transfer are crucial for efficient GPU-based spatial joins. Lastly, GPUs from specific vendors, such as NVIDIA, often come with unique hardware features, such as ray-tracing cores, that can be utilized to accelerate particular spatial operations. This stands in contrast to the more general approach of traditional CPU-based algorithms, highlighting the need for novel algorithms that can take advantage of specialized hardware features.

Existing methods for performing spatial joins on GPUs can be generally classified into three categories: (1) General-purpose GPU programming (GPGPU) methods, which rely on programming APIs like CUDA and OpenCL; (2) Rasterization-based methods, which leverage the GPU's rendering pipeline, particularly the rasterization operation, to convert geometric primitives into pixel-based representations for efficient spatial joins; and (3) Ray-tracing (RT) core methods, which utilize dedicated ray-tracing hardware in modern GPUs to calculate intersections and proximity relationships among spatial objects efficiently. This chapter focuses on using rasterization to perform spatial joins and subsequent aggregations on the GPU.

2 Historical Background

The first GPU, the GeForce 256, was introduced in 1999. Initially designed as a specialized processor for accelerating graphics rendering in 3D games, GPUs soon attracted interest for general-purpose computation. Early efforts leveraged OpenGL-based[Shreiner et al.(2013)] graphics APIs to perform spatial selection and joins [Sun et al.(2003)]. Specifically, Sun et al. explored GPU acceleration for the refinement step of spatial queries by rendering spatial geometries and analyzing pixel data to detect intersections or measure distances. For spatial intersection joins, they utilized rasterization in the join refinement phase to determine whether two polygons do not intersected. However, their approach relied on pairwise comparisons, which did not scale well with an increasing number of polygons, yielding only modest

performance improvements over traditional CPU-based methods. Moreover, early GPUs had limited programmability and computational power, which constrained their effectiveness for general-purpose spatial processing.

The emergence of *General-Purpose GPU (GPGPU)* technologies in 2007 marked a significant shift, enabling more flexible use of GPUs for non-graphics computations. This development sparked interest in employing GPUs for relational data management, which eventually extended to spatial data. Early efforts focused on accelerating large-scale Point-in-Polygon (PIP) tests for spatial joins [Zhang and You(2012)], soon followed by more advanced join methods that introduced improved filtering and load-balanced refinement on the GPU [Aghajarian et al.(2016), Aghajarian and Prasad(2017)]. Building on these techniques, subsequent research accelerated geometric intersection on GPUs using a collection of filters applied hierarchically [Liu et al.(2019), Liu and Puri(2020)]. Finally, HEAVY.AI [HEAVY.AI([n. d.])] (formerly OmniSci), a geospatial analytics platform, accelerates spatial queries by compiling them to native GPU code and leveraging GPU parallelism.

In 2017, researchers revisited GPU-based rendering for spatial query processing, leveraging the significant advancements in GPU technology since the early work of Sun et al. [Sun et al.(2003)]. Specifically, the Raster Join approach [Tzirita Zacharatou et al.(2017)] reformulates aggregate spatial join queries as a sequence of drawing operations on a canvas. This transformation enables spatial joins to be performed as intersections of rendered objects, effectively utilizing the GPU's graphics pipeline to achieve interactive speeds. Generalizing on the above approach, Spade [Doraiswamy and Freire(2022)] is a spatial database engine that leverages the GPU's graphics pipeline to implement spatial operators. It adopts the canvas data model, where spatial objects are represented as images, with each pixel storing metadata about the objects' geometry. To efficiently support a diverse range of query types, Spade integrates a GPU-friendly spatial algebra derived from computer graphics operations, taking advantage of the inherent optimization of the GPU for such operations. By executing spatial operators within the graphics pipeline, Spade fully exploits the computational power of modern GPUs.

Recently, the focus has shifted to leveraging dedicated *ray-tracing (RT)* cores in modern GPUs for spatial data processing. Ray tracing, traditionally used in gaming and rendering, simulates light interactions with geometric primitives to produce highly realistic images. It operates by tracing rays from a camera through pixels in an image plane and computing ray-primitive intersections, a computationally expensive process. To accelerate intersection detection, Bounding Volume Hierarchies (BVHs) are used to hierarchically partition the scene, significantly reducing the number of intersection tests by filtering out unnecessary primitives. The RayJoin framework [Geng et al.(2024)] transforms spatial join queries into ray-tracing problems, utilizing the hardware-accelerated BVH traversal of RT cores to efficiently identify intersecting objects within a scene, thereby significantly outperforming traditional CPU-based spatial join methods.

3 Scientific Fundamentals

This section begins by introducing the graphics pipeline, the foundation of rasterization-based approaches. It then presents Raster Join [Tzirita Zacharatou et al.(2017)], a state-of-the-art method that leverages GPU rasterization to perform spatial aggregate join queries over polygons efficiently.

3.1 Graphics Pipeline

The graphics pipeline is designed to maximize GPU performance for applications such as real-time rendering in video games. This is achieved by decomposing the complex rendering process, which transforms a 3D scene into a 2D image from the camera's perspective, into a structured sequence of specialized stages. Each stage is executed as a collection of parallel threads, allowing an efficient distribution of workload across the GPU's computational cores. This structured decomposition enables the graphics driver to dynamically schedule threads across different pipeline stages, optimizing resource utilization and minimizing execution bottlenecks. Furthermore, since the graphics pipeline is intrinsically aligned with the GPU hardware architecture, applications that leverage it inherently benefit from hardware-aware execution, ensuring efficient performance without requiring extensive manual optimization. The remainder of this section provides a brief overview of the graphics pipeline stages.

Vertex Shader. The graphics pipeline begins with the vertex shading stage, where all vertex coordinates are transformed into a unified world coordinate system before being projected onto the screen. This stage also manages additional per-vertex computations, such as lighting and texture coordinate setup. The vertex shader, executed in a Single Program Multiple Data (SPMD) fashion, allows developers to customize transformations and other per-vertex operations.

Clipping and Rasterization. After vertex processing, primitives are passed to the clipping and rasterization stage, which is typically managed by the GPU driver. Clipping removes primitives outside the visible viewport, while those partially intersecting the viewport are cropped, generating new primitives that are fully contained within the screen space. Rasterization then converts each primitive into a set of fragments, where each fragment corresponds to a potential pixel in the final image. The number of generated fragments is resolution-dependent—higher resolutions (e.g., 1920×1080) produce a greater number of smaller fragments compared to lower resolutions (e.g., 800×600).

Fragment Shader. Once rasterization is complete, the fragment shader processes each fragment. This stage applies interpolated attributes, such as color, depth, and texture effects, determining the final visual output of each pixel. The fragment shader is also programmable, allowing developers to implement advanced shading techniques. Similar to vertex shaders, fragment shaders operate in an SPMD model, leveraging the GPU's massively parallel architecture for efficient execution.

Post-Fragment Processing. Following fragment shading, the final stage of the pipeline processes the output fragments to generate the actual pixels displayed on the screen. Multiple fragments may contribute to a single pixel; these fragments undergo blending, a process that combines color information based on predefined blending functions. Additional operations, such as depth testing, stencil testing, and alpha compositing, refine the final rendered image before it is written to the frame buffer.

Frame Buffer Object (FBO). Beyond direct rendering to a physical display, OpenGL supports rendering to off-screen buffers, known as Frame Buffer Objects (FBOs). An FBO functions as a virtual rendering target, allowing developers to define arbitrary resolutions independent of the display's native resolution. Each pixel in an FBO typically stores four 32-bit values representing the red, green, blue, and alpha (RGBA) color channels, but the frame buffer can also be configured to store additional attributes, such as depth values or auxiliary render targets for advanced rendering techniques.

Overall, the graphics pipeline is designed to maximize performance through parallel processing and specialized hardware optimizations. OpenGL provides a high-level abstraction for interacting with the GPU, while graphics drivers handle low-level tasks, including thread scheduling, parallel rasterization, and memory management. Since these drivers are hardware-specific, applications utilizing the graphics pipeline inherently benefit from hardware-level optimizations, ensuring efficient execution across different GPU architectures.

3.2 Raster Join

Raster Join is a spatial data processing technique designed to efficiently compute spatial aggregate join queries involving point and polygon datasets. Unlike traditional spatial join approaches that rely on Point-in-Polygon (PIP) tests, Raster Join leverages rasterization to map geometric entities onto a discrete pixel-based representation. This technique significantly reduces computational complexity by performing spatial joins in image space, enabling high-performance execution on modern GPUs.

The design of Raster Join builds on two key observations:

- A spatial join between points and polygons can be approximated by their intersections when rendered onto a common rasterized canvas.
- Aggregation can be integrated within the join operation, eliminating the need to materialize intermediate results.

Framing spatial aggregate join queries as rendering tasks allows for the effective use of GPU optimizations, particularly for rasterization. GPU rasterization is a hardware-accelerated process that efficiently converts polygons into pixel-based representations. Since rasterization is built into the GPU driver and optimized for the underlying hardware, it maximizes GPU utilization and ensures high-performance execution.

Integrating aggregation directly within the join operation provides two major benefits: (1) it eliminates the need to store intermediate join results, allowing the GPU to process larger datasets with fewer passes; and (2) it avoids materialization and minimizes data transfer overhead, significantly improving execution speed.

3.2.1 Core Approach

The design of Raster Join builds on the aforementioned key observations. Intuitively, it first *draws* the points on a canvas and keeps track of the intersections by maintaining *partial aggregates* in the canvas cells. It then *draws* the polygons on the same canvas, and computes the aggregate result from the partial aggregates of the cells that intersect with each polygon. The above operations are accomplished in two steps as described next.

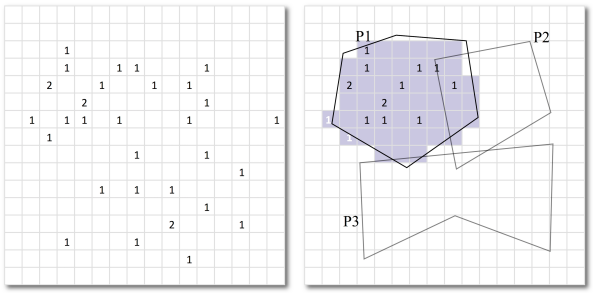

Fig. 1 Raster Join first renders all points onto an FBO storing the count of points in each pixel (a). It then aggregates the values of pixels corresponding to polygon fragments (b).

Step 1. Render Points: The input point set is transformed into screen space and rendered onto an FBO. In this FBO, Raster Join encodes the partial aggregate (e.g., count, sum) of all points falling within each pixel using the color channel of the pixel. For example, when computing a count aggregate, the red channel of the corresponding pixel is incremented by 1 for each point mapped to it. This results in an FBO where each pixel stores the aggregated value of the points it contains. Figure 1(a) illustrates this process for an example dataset using the count aggregate.

Step 2. Render Polygons: In this step (Figure 1(b)), the query result is incrementally updated. To do so, Raster Join maintains a result array A, where each entry corresponds to a polygon, initially set to zero. Similarly to the previous step, the polygon's vertices are first transformed into screen space. The transformed polygons are then converted into discrete fragments by the rasterization process. Each polygon fragment is processed as follows: the partial aggregate of points in the corresponding pixel is retrieved from the FBO from the previous step. This value is then used to

update the polygon's aggregate in the corresponding entry in the result array A. After all polygons have been processed, the array A contains the final result of the query.

3.2.2 Bounding Errors

Since Raster Join relies on rasterization, it introduces small errors due to the discrete nature of pixels. These errors arise from how pixels intersect with polygon boundaries:

False Positives: False positives occur when a pixel is classified as belonging to a polygon despite not being fully enclosed by its boundaries. This happens because rasterization assigns entire pixels to a polygon if their centers or a sufficient portion of their area falls within the polygon. Consequently, points contained in these pixels are aggregated into the polygon, even if they are technically outside the precise polygon boundary. In the example of Figure 1(b), the false positive counts are highlighted in white.

False Negatives: False negatives occur when a point should be included in a polygon's aggregation but is excluded due to pixel discretization. This happens when a pixel is not assigned to a polygon because its center does not fall within the polygon's boundaries, even though portions of the pixel intersect the polygon. As a result, points contained in such pixels can be incorrectly omitted from the polygon's aggregate.

To control errors, the resolution of the rasterized canvas is dynamically adjusted. Increasing the resolution decreases pixel size, enhancing accuracy. When the required resolution exceeds GPU capabilities, the image space is divided into smaller canvases, each processed sequentially. Formally, accuracy is regulated by imposing an error bound on the Hausdorff distance between the original polygons and their pixel-based approximations. This approach aligns well with real-world data, which often contains inherent uncertainties. For instance, neighborhood boundaries typically follow street segments, where the entire street surface—not just a thin line—is considered to be the boundary. In such cases, the street width can serve as a meaningful error bound. Additionally, in visualization applications, minor approximations often remain imperceptible, making small errors in spatial aggregation negligible for practical purposes.

3.2.3 Achieving Accurate Results

To obtain exact results, Raster Join can be enhanced with selective Point-in-Polygon (PIP) tests. Instead of performing PIP tests for all points, Raster Join only applies PIP tests to correct errors along polygon boundaries. The process involves three steps:

Step 1. Render Polygon Outlines: The first step identifies pixels that contain polygon boundaries. To achieve this, the polygon outlines are rasterized onto a separate FBO, creating a boundary mask. This mask highlights pixels where approx-

imation errors are likely to occur, ensuring that subsequent computations focus only on relevant areas.

Step 2. Render Points and Apply PIP Tests: The second step processes the input points. Each point is transformed into screen space and checked against the boundary mask. If a point falls within a boundary pixel, it undergoes an explicit Point-in-Polygon (PIP) test to determine its exact inclusion status. This selective testing prevents unnecessary computations while ensuring accuracy.

Step 3. Render Polygons: The final step simply renders all polygons, similar to the core Raster Join approach. However, when processing each fragment, the algorithm first checks if it falls on a boundary pixel. If it does, then it is discarded, since all points falling into that pixel have already been processed in the previous step. Otherwise, all points falling into the pixel are inside the polygon and are included in the final aggregate result.

This method balances performance and accuracy by leveraging rasterization for most computations while applying computationally expensive PIP tests only where necessary.

3.2.4 Extensions and Implementation

Raster join can be extended to support more complex spatial aggregate join queries and large datasets. It can also provide result ranges for each polygon, rather than just a single aggregate result.

Aggregations: In addition to simple counts, Raster Join can store auxiliary attributes in the FBO's color channels. This enables more complex aggregations, such as (weighted) sums and averages, allowing for richer analytical capabilities without significant additional computational overhead.

Filtering: When spatial aggregate join queries include constraints, such as filtering points based on specific attributes, these can be efficiently processed on the GPU. The vertex shader evaluates each data point against the query constraints before transforming it into screen space. Points that do not meet the constraints are assigned positions outside of the screen space, thereby ensuring they are clipped and excluded from further processing in the fragment shader. Currently, the supported constraints include the operators greater than (>), less than (<), and equal to (=).

Estimating Result Ranges: To improve the accuracy assessment, Raster Join can estimate result ranges by leveraging boundary pixels. Specifically, for each polygon, the algorithm can compute an upper and lower bound on the aggregate by considering the maximum and minimum possible contributions from these boundary pixels. By assuming a uniform spatial distribution of points within the boundary pixels, we can derive a confidence interval for the aggregation result. This approach is particularly useful in scenarios where precision needs to be balanced with computational efficiency.

Out-of-Core Execution: When processing point datasets that exceed GPU memory capacity, Raster Join can operate in an out-of-core manner by partitioning the data into smaller batches that fit into the GPU memory and processing them sequentially.

Implementation: Raster Join is implemented in OpenGL [Shreiner et al.(2013)] by customizing specific stages of the graphics pipeline using GLSL shaders.

4 Key Applications

Spatial aggregate join queries play a crucial role in various scientific and analytical domains by enabling efficient spatial data integration and summarization:

- **Geographic Information Systems (GIS) Applications:** GIS applications manage spatial datasets organized into distinct map layers, such as buildings, road networks, and administrative boundaries. Analyzing relationships between these layers often requires spatial aggregate join queries to associate point-based attribute data with polygonal regions. For example, to analyze business distribution, individual store locations (points) can be aggregated within commercial zones (polygons) to assess regional economic activity. Similarly, to assess traffic impact, vehicle GPS trajectories (points) can be joined with zoning areas (polygons) to compute congestion metrics. Since spatial joins are computationally intensive, optimizing their execution is critical for ensuring the efficiency of GIS applications.
- **Visualization and Interactive Analytics:** Spatial aggregate join queries are essential for real-time visualization tools that enable users to explore large-scale geospatial data interactively [Doraiswamy et al.(2018)]. By aggregating high-density point data, such as rideshare pickups, air quality sensor readings, or mobile phone activity, into geographic regions, these queries allow visualization platforms to generate heatmaps, choropleth maps, parallel coordinate charts, and other dynamic visual representations. Optimizing spatial joins in these applications is critical for maintaining interactivity and responsiveness when visualizing large datasets.
- **Urban Planning:** By integrating demographic, socioeconomic, and infrastructure datasets with administrative boundaries, spatial aggregate joins aid in zoning analysis, resource allocation, and optimization of transportation networks. Urban planners often adjust zoning boundaries and policies, requiring real-time feedback on how these changes affect essential urban metrics. Spatial aggregate joins allow for the dynamic computation of summary statistics (e.g., population density, land use distribution) within the updated zones, enabling data-driven decision-making. Additionally, urban planners can incorporate new infrastructure elements, such as bus stops or police stations, and determine their spatial coverage using restricted Voronoi diagrams. They can then aggregate urban data in the affected regions. Overall, the efficient execution of spatial aggregate join queries is crucial to maintaining interactivity in urban planning tools.
- **Environmental Modeling:** Spatial aggregate join queries facilitate large-scale ecological assessments by aggregating environmental sensor data within geographic regions. This supports various applications, including climate change studies, pollution tracking, and habitat conservation planning.

5 Future Directions

Future research can drive further advancements at the intersection of computer graphics, spatial databases, and high-performance computing. As spatial and spatio-temporal datasets continue to expand in volume and complexity, leveraging modern GPU architectures will be critical for developing scalable, high-performance solutions. One of the primary performance bottlenecks in GPU-accelerated spatial joins is data transfer between CPU and GPU memory. This overhead is primarily due to the latency and bandwidth limitations of the PCI Express (PCIe) bus, particularly when performing frequent, small data transfers. Optimizing memory management strategies could significantly enhance performance. Another important research direction is the development of cost models that accurately predict the execution costs of different GPU-based spatial join strategies by considering data, workload, and hardware characteristics. Integrating these models into query optimizers would enable spatial databases to dynamically choose the most efficient execution strategy for a given spatial join, leading to improved query planning and overall performance. Currently, most GPU-accelerated spatial join algorithms focus on two-dimensional (2D) datasets. However, an increasing number of applications require 3D spatial joins (e.g., urban modeling, neuroscience [Tzirita Zacharatou et al.(2015)]) and spatio-temporal joins (e.g., tracking moving objects). Future research could explore how ray casting, ray tracing, and collision detection techniques—commonly used in computer graphics—can be adapted to efficiently process 3D spatial relationships and temporal dependencies in large-scale datasets. To further enhance the impact of GPU-accelerated spatial (aggregate) joins, future work could focus on seamlessly integrating these techniques with spatial databases and interactive visualization tools. This integration would empower users with real-time geospatial analytics capabilities, allowing them to interactively explore large spatial datasets and visualize query results with minimal delay. Such work would promote the broader adoption of spatial (aggregate) joins in various real-world applications, including urban planning and environmental modeling.

6 Cross References

- Graphics (Processing Units) for Spatial Processing
- GPU-based Filter and Refine Algorithms for Polygon Intersection
- Cuda/GPU

References

[Aghajarian and Prasad(2017)] Danial Aghajarian and Sushil K. Prasad. 2017. A Spatial Join Algorithm Based on a Non-uniform Grid Technique over GPGPU. In *25th International Con-*

ference on Advances in Geographic Information Systems (SIGSPATIAL '17). ACM, New York, NY, USA, 56:1–56:4.

[Aghajarian et al.(2016)] Danial Aghajarian, Satish Puri, and Sushil Prasad. 2016. GCMF: An Efficient End-to-End Spatial Join System over Large Polygonal Datasets on GPGPU Platform. In *24th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '16)*. ACM, New York, NY, USA, 1–10.

[Doraiswamy and Freire(2022)] Harish Doraiswamy and Juliana Freire. 2022. GPU-Powered Spatial Database Engine for Commodity Hardware: Extended Version. *arXiv preprint arXiv:2203.14362* (2022). https://arxiv.org/abs/2203.14362

[Doraiswamy et al.(2018)] Harish Doraiswamy, Eleni Tzirita Zacharatou, Fabio Miranda, Marcos Lage, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. 2018. Interactive Visual Exploration of Spatio-Temporal Urban Data Sets using Urbane. In *44th International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1693–1696.

[Geng et al.(2024)] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *38th International Conference on Supercomputing (ICS '24)*. ACM, New York, NY, USA, 124–136.

[HEAVY.AI([n. d.])] HEAVY.AI. [n. d.]. HEAVY.AI: GPU-Accelerated Analytics Platform. https://www.heavy.ai/. Accessed: 2025-02-11.

[Liu and Puri(2020)] Yiming Liu and Satish Puri. 2020. Efficient Filters for Geometric Intersection Computations using GPU. In *28th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '20)*. ACM, New York, NY, USA, 487–496.

[Liu et al.(2019)] Yiming Liu, Jie Yang, and Satish Puri. 2019. Hierarchical Filter and Refinement System over Large Polygonal Datasets on CPU-GPU. In *26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 141–151.

[Shreiner et al.(2013)] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3* (8th ed.). Addison-Wesley Professional.

[Sun et al.(2003)] C. Sun, D. Agrawal, et al. 2003. Hardware Acceleration for Spatial Selections and Joins. In *2003 International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 455–466.

[Tzirita Zacharatou et al.(2017)] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Claudio T. Silva, and Juliana Freire. 2017. GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. *Proceedings of the VLDB Endowment* 11, 3 (2017), 352–365.

[Tzirita Zacharatou et al.(2021)] Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Varun Pandey, Harish Doraiswamy, and Volker Markl. 2021. The Case for Distance-Bounded Spatial Approximations. In *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*. 7 pages.

[Tzirita Zacharatou et al.(2015)] Eleni Tzirita Zacharatou, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2015. RUBIK: efficient threshold queries on massive time series. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management* (La Jolla, California) *(SSDBM '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 12 pages. https://doi.org/10.1145/2791347.2791372

[Zhang and You(2012)] Jianting Zhang and Simin You. 2012. Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. In *20th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '12)*. ACM, New York, NY, USA, 104–113.