SwiftSpatial: Spatial Joins on Modern Hardware

WENQI JIANG, Systems Group, ETH Zurich, Switzerland
OLEH-YEVHEN KHAVRONA, Systems Group, ETH Zurich, Switzerland
MARTIN PARVANOV, Systems Group, ETH Zurich, Switzerland
GUSTAVO ALONSO, Systems Group, ETH Zurich, Switzerland

Spatial joins are among the most time-consuming spatial queries, remaining costly even in parallel and distributed systems. In this paper, we explore hardware acceleration for spatial joins by proposing *SwiftSpatial*, an FPGA-based accelerator that can be deployed in data centers and at the edge. SwiftSpatial contains multiple high-performance join units with innovative hybrid parallelism, several efficient memory management units, and an extensible on-chip join scheduler that supports the popular R-tree synchronous traversal and partition-based spatial-merge (PBSM) algorithms. Benchmarked against various CPU and GPU-based spatial data processing systems, SwiftSpatial demonstrates a latency reduction of up to 41.03× relative to the best-performing baseline, while requiring 6.16× less power. The performance and energy efficiency of SwiftSpatial demonstrate its potential to be used in a variety of configurations (e.g., as an accelerator, near storage, in-network) as well as on different devices (e.g., data centers where FPGAs are widely available or mobile devices, which also contain FPGAs for specialized processing).

CCS Concepts: • **Information systems** → **Data management systems**; **Spatial-temporal systems**; • **Computer systems organization** → **Parallel architectures**.

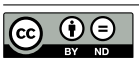
Additional Key Words and Phrases: Spatial Data; Spatial Query Processing; Hardware Acceleration; FPGA

ACM Reference Format:
Wenqi Jiang, Oleh-Yevhen Khavrona, Martin Parvanov, and Gustavo Alonso. 2025. SwiftSpatial: Spatial Joins on Modern Hardware. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 224 (June 2025), 27 pages. https://doi.org/10.1145/3725361

1 Introduction

Spatial joins — where two sets of spatial objects are joined based on their spatial relationships — are time-consuming spatial queries even if many parallel and distributed spatial data processing systems have been developed [10, 23, 56]. For instance, a 2013 study on spatial big data frameworks [10] reported that joining two indexed datasets, totaling fewer than 100 million records, required 10 minutes (equivalent to 32 CPU hours) when executed on eight servers equipped with 192 cores and 1TB of memory in total. Such performance has become insufficient considering the increasing number of moving objects, such as autonomous vehicles and IoT devices [63, 66]. After decades of extensive research on efficient indexes and algorithms for spatial joins, further performance gains have become increasingly difficult given a certain computational budget. This prompts the question: *is it possible to improve spatial join performance without demanding more resources such as CPU cores or memory capacity?*

Authors' Contact Information: Wenqi Jiang, Systems Group, ETH Zurich, Zurich, Switzerland, wenqi.jiang@inf.ethz.ch; Oleh-Yevhen Khavrona, Systems Group, ETH Zurich, Zurich, Switzerland, yevhen.khavrona@inf.ethz.ch; Martin Parvanov, Systems Group, ETH Zurich, Zurich, Switzerland, martin.ivov.parvanov@gmail.com; Gustavo Alonso, Systems Group, ETH Zurich, Zurich, Switzerland, alonso@inf.ethz.ch.



ACM 2836-6573/2025/6-ART224
https://doi.org/10.1145/3725361

Given the increasing computational demands of spatial joins and the slowdown in Moore's law, we approach spatial joins from a hardware acceleration perspective in this paper. Specifically, we choose FPGAs as the hardware platform for the designed accelerator due to their growing adoption in the cloud, as evidenced by deployments at Microsoft [26], Amazon [2], or Alibaba [49].

Spatial join algorithms generally follow the following paradigm. During the preprocessing phase, both input datasets are decomposed into small tiles. For example, the datasets can be partitioned using a simple grid [57, 69] or indexed by a tree [30, 52, 62]. After preprocessing the datasets, the join process begins, which comprises two primary components. Firstly, a control flow identifies tile pairs from both datasets that could potentially produce results. For example, when both datasets are partitioned using a grid, the control flow selects identical tiles from both datasets; for datasets indexed by R-trees, synchronous traversal [13] is adopted to navigate both trees to identify intersected tile pairs. Secondly, for each candidate tile pair, all objects in the tiles are evaluated based on the predicate to produce results or intermediate outcomes. The repetitive nature of these operations suggests that they could be amenable to parallelization using parallel processing units and pipeline parallelism, both available on FPGAs.

Thus, we propose *SwiftSpatial*, an in-memory FPGA-based accelerator designed for high-performance and energy-efficient spatial joins. SwiftSpatial supports two widely used spatial join algorithms: R-tree synchronous traversal [13] and partition-based spatial-merge (PBSM) [57], and can be extended to support other spatial join algorithms due to its modular design. With R-tree synchronous traversal, SwiftSpatial can leverage the R-trees already maintained by various spatial data processing systems, such as PostGIS [8], Oracle Spatial [47], Apache Sedona [1], and SpatialSpark [73]. The support for PBSM is motivated by its scalability on parallel architectures, as already shown on multi-core CPUs [69].

The key idea of SwiftSpatial is to build efficient hardware primitives for spatial joins by instantiating multiple specialized join units. (C1) Each SwiftSpatial join unit is optimized to achieve minimal latency when joining two spatial object tiles. Its superior performance stems from the innovative architecture that employs hybrid parallelism : (a) operator parallelism for fast predicate evaluation involving multiple object boundary comparisons and (b) pipeline parallelism between input ingestion, join predicate evaluation, and result production for qualifying object pairs. The hybrid parallelism allows a join unit to process an object pair every clock cycle, leading to high and predictable tile-level join performance. Multiple join units can be instantiated to parallelize joins across different tiles. **(C2)** Moreover, SwiftSpatial incorporates an on-chip scheduler to orchestrate these join units, passing control signals directly to the join units, thus supporting complex control flow with minimal performance overhead. **(C3)** In addition to the join and control units, SwiftSpatial features specialized memory management units to further improve join performance. By directly managing the physical address space, the memory management units (a) optimize memory access patterns through memory request batching and memory address rewriting, and (b) avoid the overhead associated with page table management and dynamic memory allocation prevalent in software systems.

We evaluate SwiftSpatial on real-world and synthetic datasets of various size scales and compare it against a variety of CPU and GPU-based spatial data processing systems, including PostGIS [8], Apache Sedona [1], SpatialSpark [73], cuSpatial [6], and our multi-threaded C++ implementations of R-tree synchronous traversal and PBSM. Overall, SwiftSpatial demonstrates outstanding performance and energy efficiency, achieving up to 41.03× latency reduction compared to the best-performing baseline system while consuming only 23.48W of power, 6.16× less than the baseline CPU. Moreover, the modular design of SwiftSpatial suggests it can not only be instantiated on data-center-grade FPGAs but also has the potential for deployment on embedded systems, showing its versatility and wide-ranging applicability.

The contributions of the paper are as follows:

- We design SwiftSpatial, a spatial join accelerator with:
  - Spatial join units with innovative hybrid parallelism.
  - Multiple efficient memory management units.
  - An on-chip spatial join scheduler that coordinates join units and memory management units.
- We implement SwiftSpatial on FPGA and support R-tree synchronous traversal and PBSM algorithms as the control flows.
- We compare SwiftSpatial with CPU and GPU-based spatial data processing systems on various datasets, showcasing its performance and energy efficiency.

2 Background and Motivation

2.1 Spatial Join

Let two datasets $R \subset \mathbb{R}^d$ and $S \subset \mathbb{R}^d$ consist of spatial objects in a d-dimensional space. A spatial join between R and S returns a set of all object pairs that meet a specific join predicate, such as *intersects* or *contains*. Equation 1 defines spatial join based on the *intersects* predicate. In practice, spatial joins are typically carried out in two or three-dimensional spaces, in which each object can have an arbitrary geometry, such as a point, circle, rectangle, etc.

$$R \bowtie_{\cap} S = \{(r, s) | r \in R, s \in S, r \cap s \neq \emptyset\} \tag{1}$$

A spatial join involves two steps: *filtering* and *refinement*, and this paper focuses on filtering.

Filtering. The filtering step aims to quickly identify potential candidate pairs of geometries that may satisfy the spatial predicate. To achieve this, the filtering step typically relies on either indexes like R-trees [30] and quadtrees [61] or data partitioning to prune non-relevant geometries efficiently and (b) approximate geometries with simpler structures such as *minimum bounding rectangles (MBRs)*. The filtering step returns the set of all object pairs from the two input sets whose MBRs intersect. This step may yield some false positives that satisfy the join predicate on approximated geometries but actually do not upon closer examination.

Refinement. The refinement step takes the results from the filtering step and verifies the actual spatial relationship between the candidate pairs [12, 27, 80]. To eliminate the false positives from the filtering step, this phase determines whether the pairs satisfy the given join predicate (e.g., intersects, contains, etc.) using the actual geometries instead of the MBRs.

2.2 R-tree and Synchronous Traversal

Extensive research has been conducted on spatial join algorithms. One category of approaches indexes or partitions one dataset and iterates over all objects of the other dataset as queries. Some representative indexes in this category include the R-tree [30], linearized KD-Trie [54], and simple grid [64]. Another category of approaches simultaneously evaluates both datasets. These approaches include plane-sweep [19], partition-based spatial-merge join (PBSM) [57], and R-tree synchronous traversal [13]. In this section, we delve into R-tree synchronous traversal due to its great performance [66].

As visualized in Figure 1, an R-tree is a balanced tree structure. The tree has two types of nodes: directory nodes and leaf nodes. A directory node stores the MBRs and pointers to its child nodes, while a leaf node contains the MBRs and IDs of the spatial objects within the database.

An R-tree satisfies the following properties, considering that m and M represent the minimum and the maximum entries per node, respectively ($2 \leq m \leq \lceil M/2 \rceil$). First, all nodes, excluding the root, contain between m and M entries. Second, the root node may hold only one entry if the root

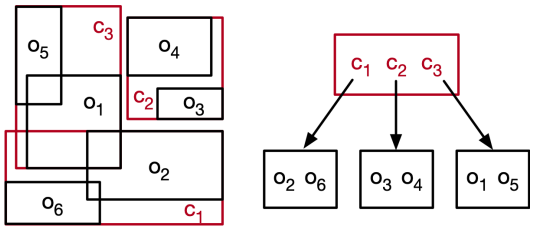


Fig. 1. A two-level R-tree index containing six objects.

Algorithm 1: Synchronous Traversal

input : RTree_R, RTree_S
output: Join results $\{(r, s) | r \in R, s \in S, r \cap s \neq \emptyset\}$
return *RecursiveTraverse(*$RTree_R$*.root,* $RTree_S$*.root))*

is a leaf node and there is only one object in the dataset, while the upper bound M still applies. Third, all the leaf nodes have the same distance to the root node.

An R-tree can be constructed dynamically or statically. The original R-tree paper [30] manages tree construction by progressively inserting new objects into the tree. The R*-tree [11] enhances tree quality through more intricate insertion strategies. The advantage of dynamic insertion is its ability to intermix with queries and deletions. An alternative approach is to bulk-load the entire dataset to construct the R-tree, and the most widely-used approaches are Sort-Tile-Recursive (STR) [48] and Hilbert R-tree [41]. Bulk-loading produces superior R-tree topologies compared to dynamically constructed R-trees, improving query performance.

R-tree facilitates efficient spatial query processing by exploiting the hierarchical structure and MBRs to prune the search space. For instance, a window query on R-trees using an intersection predicate seeks to return all objects that intersect with the query geometry. R-tree handles such a query by starting the search from the root node and traversing the tree down to the leaf nodes. Given a directory node on the search path, all the MBRs of its children are evaluated to determine whether they intersect the query window. Only when an intersection is found will the traversal continue to explore the corresponding subtree. The objects on the searched leaf nodes are compared with the query window, and the qualified ones are returned as results.

Although a spatial join can be implemented by querying a dataset indexed by an R-tree with many window queries representing the objects from the other dataset, synchronous traversal using R-trees constructed on both datasets can yield better performance [13, 66].

Synchronous traversal is an efficient spatial join algorithm [13]. It refers to the simultaneous traversal of two R-trees, one for each dataset being joined. The goal is to identify object pairs that potentially satisfy the join predicate by exploiting (a) the hierarchical structure of both R-trees and (b) MBRs associated with each R-tree node, such that the search space is pruned effectively.

Algorithms 1 and 2 summarize synchronous traversal on two polygon datasets using the intersect predicate. Starting at the root nodes of both R-trees, as shown in Algorithm 1, it proceeds in a depth-first search manner. Algorithm 2 details the procedure at each traversal step. If both inputs are leaf nodes, the object pairs satisfying the join predicate are added to the result set. If both inputs are directory nodes, the algorithm continues to explore the children pairs that satisfy the join predicate through recursive traversal of the qualified child nodes. If only one of the nodes is a leaf, the recursive traversal proceeds between the leaf node and the qualified children of the directory nodes.

Algorithm 2: Recursive Traverse

```
input: Node_R, Node_S, ResultSet
if Node_R.isLeaf and Node_S.isLeaf then
    for Obj_R ∈ Node_R, Obj_S ∈ Node_S do
        if Obj_R.MBR ∩ Obj_S.MBR then
            ResultSet.add((Obj_R, Obj_S))
else if Node_R.isDirectory and Node_S.isDirectory then
    for Child_R ∈ Node_R, Child_S ∈ Node_S do
        if Child_R.MBR ∩ Child_S.MBR then
            TraverseRecursive((Child_R, Child_S))
else
    for Child_Dir ∈ Node_Dir do
        if Node_Leaf.MBR ∩ Child_Dir.MBR then
            TraverseRecursive((Node_Leaf, Child_Dir))
return
```

Algorithm 3: Partition-Based Spatial-Merge (PBSM)

```
Input  :Dataset R, Dataset S, Grid G
/* Phase 1: Data Partition                                   */
foreach tile T_i ∈ G do
    (R_i, S_i) ← (R ∩ T_i, S ∩ T_i)
/* Phase 2: Tile-wise Join                                   */
foreach tile T_i ∈ G do
    ResultSet ← ResultSet ∪ Join(R_i, S_i);
return ResultSet
```

2.3 Partition-Based Spatial-Merge (PBSM)

Algorithm 3 shows the procedure of PBSM. It first partitions input datasets using a uniform grid. Spatial objects are then assigned to the grid tiles they intersect. For each tile, objects from one dataset are joined with those from the other. To avoid duplicate join results, a pair of intersecting rectangles is reported only if a reference point (e.g., the top-left corner) of the intersection region lies within the tile [20]. This partitioning and tile-wise join approach is well-suited for parallel architectures, as demonstrated on multi-core CPUs [69].

The plane sweep algorithm [19] is often applied for tile-wise joins in PBSM instead of the nested loop join used for R-tree node pairs, as plane sweep avoids all-to-all object comparisons. As shown in Algorithm 4, it first sorts the objects of both input datasets along one axis. A sweep line then moves across the other axis, maintaining an active set of objects that intersect with the line for each dataset. At each step, the algorithm compares the left-most object from the two datasets and inserts it into its respective active set. The algorithm then removes any objects from the opposite active set that no longer intersect with the current sweep line. Afterward, it performs intersection checks between the current object and the remaining objects in the opposite active set. This process repeats until all objects from both datasets have been processed. By efficiently managing these active sets and only comparing objects that are currently intersecting with the sweep line, the plane sweep algorithm can significantly reduce the number of comparisons compared to the nested loop join.

Algorithm 4: Plane Sweep

```
input: Dataset R, Dataset S, ResultSet
/* Step 1: Sorting                                                    */
R_sorted, S_sorted ← SortLeft(R), SortLeft(S)
/* Step 2: Sweeping through both datasets                             */
ActiveSet_R, ActiveSet_S, ResultSet ← ∅, ∅, ∅
while R_sorted.notEmpty or S_sorted.notEmpty do
    if R_sorted.first < S_sorted.first then
        ActiveSet_R.insert(R_sorted.first)
        ActiveSet_S.remove_inactive(R_sorted.first)
        ResultSet.add(ActiveSet_S.search(R_sorted.first))
        R_sorted.pop()
    else
        ActiveSet_S.insert(S_sorted.first)
        ActiveSet_R.remove_inactive(S_sorted.first)
        ResultSet.add(ActiveSet_R.search(S_sorted.first))
        S_sorted.pop()
return ResultSet
```

2.4 Existing Performance Optimizations

Over the past four decades, researchers have extensively studied spatial join algorithms, thus, further improvements by algorithm innovations have become increasingly challenging. Therefore, to enhance spatial join performance, system implementations have become increasingly important [63]. This can be achieved by either increasing system resources allocated to the join or boosting efficiency per compute unit through hardware acceleration.

Approach 1: scale-up and scale-out. Scaling up spatial join on a single server involves harnessing the power of multi-core CPUs and SIMD instructions [53, 60]. For further system performance improvements, one can use big data frameworks for spatial data to scale out spatial join operations [24, 56], such as SpatialSpark [74], LocationSpark [68], SpatialHadoop [23], HadoopGIS [10], and Simba [71]. However, neither the scale-up nor the scale-out solution improves the spatial join efficiency, as the increased performance comes at the expense of additional computing resources.

Approach 2: hardware acceleration. Researchers have been exploring the use of hardware accelerators such as GPUs to speed up spatial joins. GPUs are well-suited for some types of spatial queries, as their graphics primitives operating on polygons align closely with spatial objects [21, 22]. Specifically, the GPU can map polygons into pixels through rasterization [67, 76] and perform spatial operations, such as intersection checks, at the pixel level. This approach is particularly effective for the refinement step in the spatial join pipeline, as GPUs have significant advantages over CPUs when dealing with highly complex polygons.

However, GPUs have not been shown to be suitable for the filtering step of spatial joins, because the complex control flow in synchronous traversal does not align well with the many-core GPU architecture. Even for simpler cases of batched window or range queries on R-trees [45, 50, 58, 72], GPUs face significant challenges regardless of whether adopting BFS or DFS. In DFS, each GPU thread is responsible for a single query window, leading to significant workload imbalances between threads, thus the performance is limited by the slowest thread. In BFS, memory management becomes challenging since the number of result pairs produced by a thread block is unknown beforehand. Neither memory overflow nor reserving a huge amount of memory per thread block

is ideal. The former leads to execution failures while the latter limits the number of concurrently running thread blocks.

2.5 Field Programmable Gate Arrays (FPGAs)

In this work, we prototype our SwiftSpatial accelerator design on FPGAs, leveraging their reconfigurability and widespread availability in data centers.

Architecture. FPGAs lie between general-purpose processors (e.g., CPUs) and application-specific integrated circuits (ASICs). They behave like ASICs but can be reconfigured virtually an infinite number of times, offering both high performance and design flexibility. Developers can design custom micro-architectures tailored to specific applications, compile the design to a bitstream file representing the circuit configuration, and load the bitstream onto FPGAs to begin accelerating the applications. This process is enabled by a hardware compiler, which maps a logical design to the physical hardware units on FPGAs. These units mainly include (a) Block-RAM (BRAM) and the Xilinx-specific Ultra-RAM (URAM) as small yet fast on-chip memory, (b) Flip-Flops (FF) as registers, (c) Digital Signal Processors (DSP) as computing units, and (d) lookup-tables (LUT) as memory or computing units.

Programming. Developing FPGA accelerators typically requires significantly more effort compared to software design. Traditionally, FPGAs are programmed using hardware description languages (HDL), such as Verilog and VHDL, where developers define the circuit behavior at the granularity of a single clock cycle. Recently, High-Level Synthesis (HLS) has gained popularity, as it allows programmers to design circuits at a higher level using C/C++ or OpenCL [7, 9]. However, developing accelerators with HLS demands additional effort to learn the specific hardware-friendly coding style and to fine-tune performance by exploring a wide range of *pragmas*.

3 SwiftSpatial: Accelerator Design

We propose *SwiftSpatial*, an FPGA-based accelerator for high-performance and energy-efficient spatial join filtering. We introduce the hardware design in this section, leaving its system integration in the following section.

3.1 Design Principles

As introduced in §2, spatial join algorithms commonly decompose the join process into many sub-join tasks on tiles of objects (e.g., nodes in R-trees or tiles in PBSM). **The key idea of SwiftSpatial is to offer fast hardware primitive for joining small tiles of objects.** Consequently, an efficient accelerator for spatial joins should meet three fundamental criteria:

- Swift execution of each sub-join on a tile pair.
- Parallelization of sub-join tasks on multiple join units.
- Minimal task parallelization overhead even for fine-grained control flows like R-tree synchronous traversal.

3.2 Accelerator Overview

Reflecting the above criteria, we design *SwiftSpatial*, an in-memory spatial join accelerator for the filtering phase.

SwiftSpatial instantiates multiple specialized join units for the efficient joining of object tiles. The join units implement nested loop join rather than plane sweep for the following reasons. While plane sweep — designed to minimize the number of predicate evaluations through sorting — is a frequent choice in software solutions, this preference stems from the slow predicate evaluations on CPUs. In contrast, with our proposed architecture employing hybrid parallelism, predicate

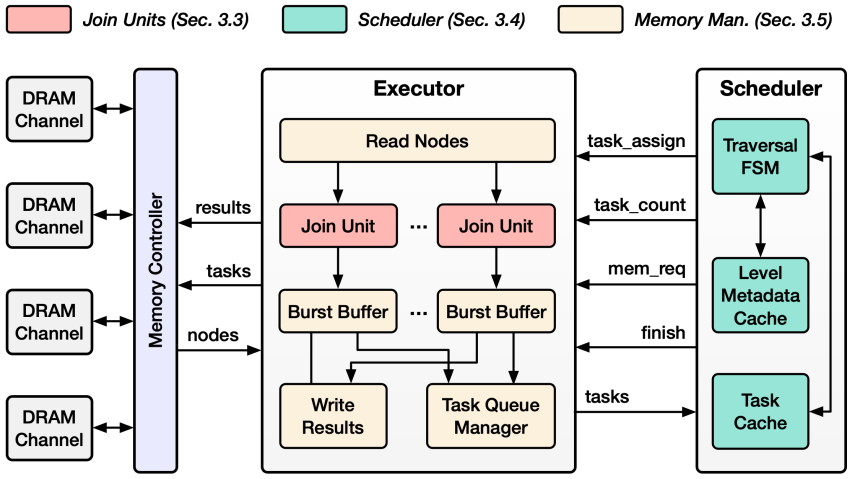


Fig. 2. SwiftSpatial accelerator overview.

evaluations between objects become notably faster, even if each evaluation involves multiple object boundary comparisons. Furthermore, the efficiency of plane sweep depends on data distributions and tile sizes. Specifically, its performance often lags behind the nested loop join on datasets with overlapped objects or when utilizing small tiles, as we will illustrate in §5. Finally, plane sweep is a sequential algorithm that benefits more from a high clock frequency than from a parallel hardware architecture.

Another key accelerator component is the on-chip scheduler that manages the join control flow and the coordination of the join units. Executing the control flow on-chip results in low performance overhead compared to the alternatives like relying on an external CPU for control signal transmission. In this paper, we not only implement PBSM with a simple control flow but also incorporate R-tree synchronous traversal into the scheduler, showcasing the efficient execution of a complex control flow in SwiftSpatial. The integration of R-tree synchronous traversal also allows SwiftSpatial to leverage the existing R-tree indexes maintained by spatial data management systems, such as PostGIS [8], Oracle Spatial [47], Apache Sedona [1], and SpatialSpark [73], avoiding additional index construction costs during spatial joins.

Figure 2 overviews SwiftSpatial. Beyond the join units and the on-chip scheduler, SwiftSpatial encompasses a set of memory management units, which optimize data transfer between the accelerator and DRAM. On-chip communication between function units, like the join units and the scheduler, is facilitated by FIFOs, akin to pipes in a software context.

3.3 Join Units

A join unit processes a pair of nodes from the two R-trees being joined and generates all intersected entry pairs.

Figure 3 shows the microarchitecture design of a join unit. The input node pairs are streamed into the join unit through two FIFOs. A join unit consists of two on-chip SRAM slices used to store the pair of input nodes. During each iteration of the join process, a pair of objects from the nodes are read from SRAM into registers. Then, multiple comparators are employed to assess whether the MBRs intersect, where all the comparators work in parallel. Specifically, there are six comparisons for 3D MBRs or four for 2D MBRs: $r.right \geq s.left$, $s.right \geq r.left$, $r.top \geq s.bottom$, $s.top \geq r.bottom$, $r.front \geq s.back$, $s.front \geq r.back$, where r and s are two entries from the pair of input

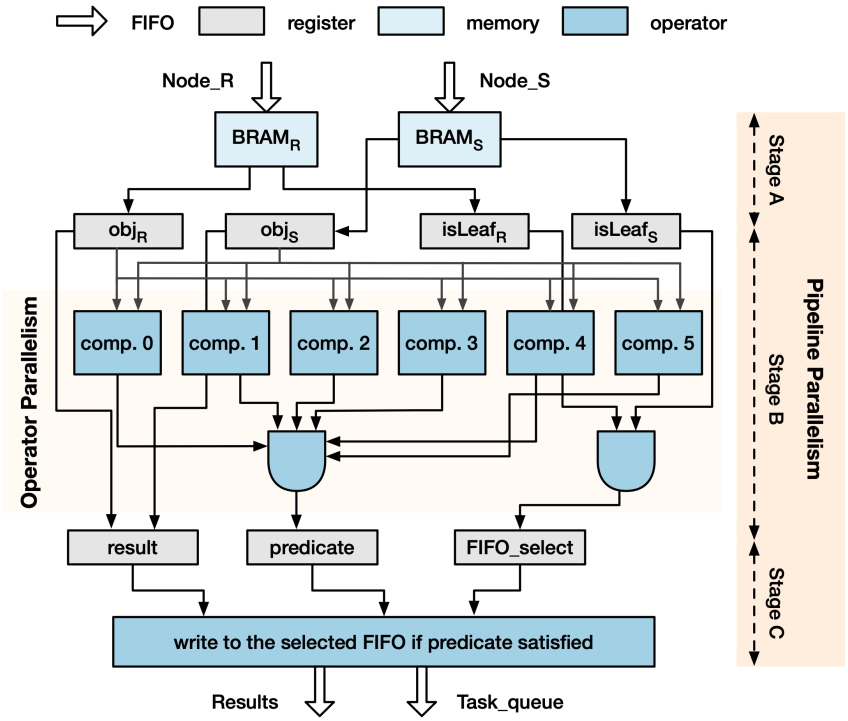


Fig. 3. The microarchitecture design of a join unit.

nodes. If all the comparison outcomes are true (evaluated by the AND gate), the pair of objects will be written to one of the output FIFOs. If both nodes are leaf nodes, the pair is written to the final result FIFO; otherwise, it is written to the task queue FIFO.

A join unit processes a pair of objects per cycle, thanks to the hybrid parallelism in the architecture. As shown in Figure 3, the first level of parallelism is operator parallelism. The combinational logic that evaluates different comparisons simultaneously and aggregates the comparison results can be completed within a clock cycle. Another form of parallelism is pipeline parallelism. Figure 3 shows different levels of registers in the pipeline. In the first stage of the pipeline, the objects are read from SRAM into the object registers. In the second stage, whether the MBRs intersect is evaluated. In the final stage, the qualified results are written into FIFOs. Thus, there are three object pairs in progress within the pipeline, allowing one pair of objects to be introduced into the pipeline every single clock cycle.

Figure 4 demonstrates the control flow of a hardware spatial join unit. The join unit follows a series of steps until it receives the finish signal. Initially, it receives node metadata from the read unit, which indicates the number of node entries and whether it is a leaf or directory node. Based on this information, the join unit reads the corresponding node data. Once the reading process is complete, the join procedure starts. During the procedure, the join unit examines every pair of entries between the node pair and checks if they intersect, outputting the qualifying pairs. If both input nodes are leaves, the outputs are recorded as results; otherwise, the generated node pairs are treated as tasks for future join operations. After joining the entries, the join unit checks if there is a finish signal sent from the scheduler, and it proceeds with the next pair of nodes if it has not received one.

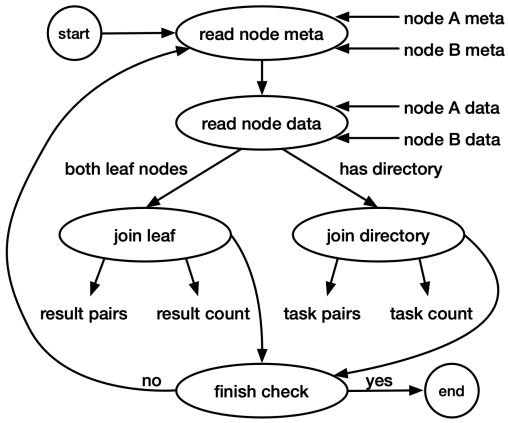


Fig. 4. The control flow of a SwiftSpatial join unit.

3.4 On-chip Scheduler

Although one approach to manage spatial join on SwiftSpatial is to combine the hardware join primitives with a software control flow for task scheduling, this method can be time-consuming, because it necessitates many accelerator kernel invocations and frequent data transfers between the host server and the accelerator.

To avoid the software scheduling overhead, SwiftSpatial incorporates an on-chip hardware scheduler to manage the control flow of spatial join and coordinate all function units, including the read and write units, join units, and the task queue manager.

3.4.1 Synchronous Traversal Scheduler. **We modify the synchronous traversal algorithm to adopt BFS, maximizing task-level parallelism among the join units.** This is essential because the original synchronous traversal, as illustrated in Algorithm 2, employs a depth-first search (DFS), which allows joining only one node pair at a time and fails to fully leverage the parallelism capabilities of SwiftSpatial. By contrast, the BFS synchronous traversal [33] performs the join level by level, such that the tasks in each level can be easily parallelized. At each level, the scheduler considers all pairs of nodes to join and assigns the tasks to the join units. For instance, at the root level, the sole pair of nodes to join is the pair of roots. The scheduler assigns the join task to the first join unit and waits for the result to be written to memory by the task queue manager. Assuming the roots are not leaf nodes, all the results generated by joining the roots become tasks for the second level, and the scheduler assigns these tasks to the join units. This process continues until all tasks at the leaf level are completed, and the results are written back to memory.

Figure 5 details the workflow of the scheduler. To start the BFS synchronous traversal in a level, the scheduler sends the physical addresses to the task queue manager to inform it where to begin writing the intermediate results (future tasks). Next, it starts dispatching jobs to the join units. To accomplish this, it requests the results from the previous level from the task queue manager. Because the task queue manager needs to read the requested task (a pair of nodes) from memory, which involves a random DRAM access, we optimize memory bandwidth utilization by implementing burst loading: reading a sequence of node pairs that can be issued next and caching them in the scheduler's SRAM. As a result, the scheduler only needs to send a request to the task queue manager when its local cache is empty. After obtaining the node pair as a task, the scheduler looks for a join unit to assign the task. Specifically, the scheduler dispatches tasks to the join units in a round-robin manner. With the node pair and join unit ID, the scheduler sends this information to the read unit,

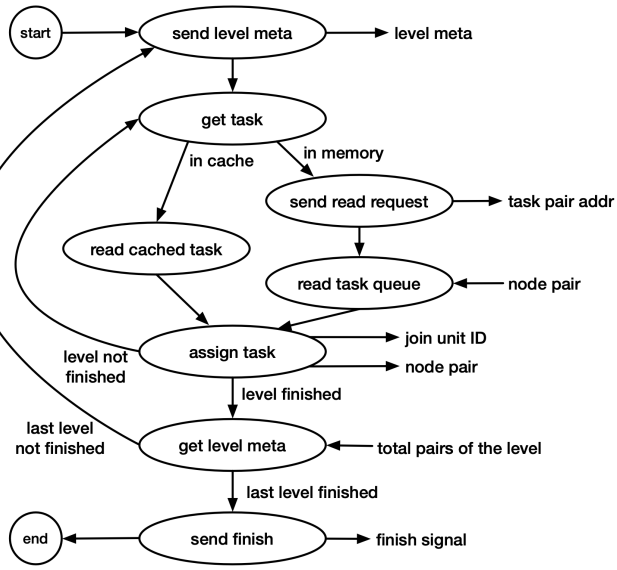


Fig. 5. The control flow of the on-chip scheduler.

which loads the node pair from memory and passes the data to the respective join unit. The task assignment continues until all the tasks on the current level are completed. The task queue manager then sends the scheduler the number of pairs produced in the current level, allowing the scheduler to determine the starting physical address to write from in the next level. The aforementioned procedures are repeated until the leaf level is completed. The scheduler finally sends a finish signal to other function units, indicating all tasks have been dispatched.

3.4.2 PBSM Scheduler. The PBSM scheduler operates similarly to the leaf-level join in synchronous traversal. Given multiple tiles to be joined, the scheduler distributes tasks across different join units. SwiftSpatial supports both static and dynamic scheduling policies. In static scheduling, tasks are predefined and assigned to specific join units, while in dynamic scheduling, an incoming task is allocated to the first available idle join unit. Empirically, these scheduling policies show similar performance, as the large number of tiles to be joined typically balances workload distribution across the join units.

We adjust the partitioning strategy on the software side to better leverage the join units. Since each join unit performs a nested loop join instead of a sweep, join performance is sensitive to tile size, as the number of object pairs to compare grows quadratically with the geometric mean of the number of objects in a tile. Thus, we set an upper bound of workload per tile by allowing hierarchical partitioning for tiles containing a large number of objects. For example, with a geometric mean tile size of 32, the number of comparisons to perform between the objects of the two datasets is limited to $32^2 = 1024$, although one dataset may contain more than 32 objects in the tile.

3.5 Memory Management

Physical address space management. SwiftSpatial directly manages the physical address space, eliminating the overhead associated with dynamic memory allocation, deallocation, and virtual address space management typically seen in CPU- and GPU-based spatial joins. For write operations, SwiftSpatial uses a self-incrementing counter to efficiently manage result writing: intermediate or final results from all join units are streamed into the write units, which concatenate the results

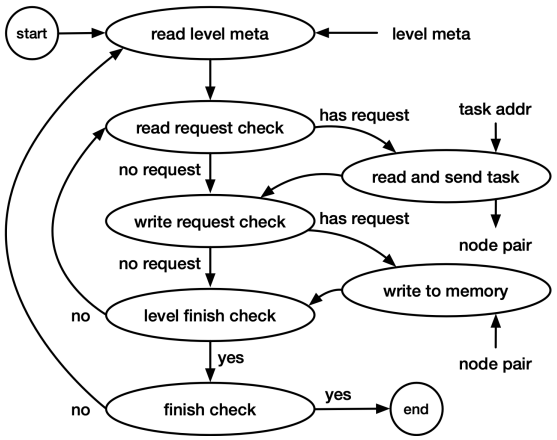


Fig. 6. The control flow of the task queue manager.

and write them to memory in bursts. This approach removes the need for individual join units to allocate memory based on the join cardinality.

Optimizing bandwidth utilization via request bursting. While reading nodes exhibit a sequential memory access pattern, the write requests in the accelerator can be highly random. This randomness arises from join operations, where, for example, it may take a join unit several cycles to output the second pair of intersected objects after the first is sent. Several join units can interleave write requests, with each write being only 8-byte long (a pair of integer object IDs).

To solve this problem, we introduce a burst buffer after every join unit. The burst buffer takes the join results as input and sends a burst of results to the task queue manager or the result write unit. The burst buffer will output a burst either when the accumulated results reach a size threshold, e.g., 4 KB, or when it is the end of joining a pair of nodes. The write result unit or the task queue manager pulls the burst results from the burst buffers in a round-robin fashion and writes the data to DRAM sequentially.

Task queue manager. According to the scheduler's requests, the task queue manager (a) writes the intermediate results, i.e., the node pairs on non-leaf levels that intersect, as future tasks to memory or (b) reads node pairs to be joined from memory.

The task queue manager workflow is illustrated in Figure 6. It starts by receiving the level metadata from the scheduler, which indicates the number of join tasks in the current level and the starting physical address for storing intermediate results. Next, it checks for any read requests from the scheduler and returns the corresponding node pairs. Subsequently, the task queue manager pulls data from the burst buffer and writes these intermediate results to memory in bursts. The process continues until the manager receives a level finish signal at the leaf level.

3.6 FPGA Implementation

We choose to develop SwiftSpatial on FPGAs for two primary reasons: (a) it facilitates direct performance evaluation of the accelerator in contrast to the simulation-based approach, and (b) the widespread adoption of FPGAs in data centers makes SwiftSpatial readily integrable with spatial data management systems [2, 26, 49, 51, 79]. We developed SwiftSpatial on FPGAs using Vitis HLS 2022.1 in C/C++ and set the accelerator's clock frequency as 200 MHz.

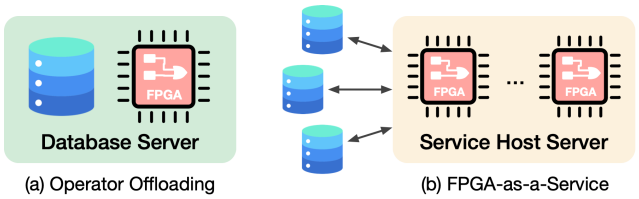


Fig. 7. Two ways of integrating SwiftSpatial into a spatial data management system.

4 System Integration

A spatial data management system could leverage SwiftSpatial in a hybrid CPU-FPGA architecture. In this architecture, the CPU constructs and maintains spatial indexes (e.g., R-trees) or partitions (e.g., PBSM) and handles non-join queries such as window queries, while the accelerator processes spatial join queries. Upon receiving a spatial join query, the up-to-date indexes are transferred to the accelerator, which then processes and returns the join results. As we will show in the experiments, the index transfer time is negligible compared to the computationally intensive join operation.

4.1 FPGA Placements in the System

There are two ways to integrate SwiftSpatial into a spatial data management system (Figure 7).

Operator offloading. In this approach, the CPU server operates a data management system while the accelerator serves as an offloading engine. An example of this offloading architecture is Alibaba's cloud-native database engine PolarDB, which accelerates LSM-trees (log-structured merge trees) by offloading compactions to FPGAs [32, 79].

FPGA-as-a-Service (FaaS). SwiftSpatial can function as a cloud-based service accessible to multiple users, avoiding the cost of integrating an accelerator per DBMS. Users can submit spatial join queries to the service, either by providing raw datasets or serialized R-trees. The host server of SwiftSpatial processes the user's requests from the network, constructs indexes if needed, forwards the queries to the accelerator, and returns the results to the users. An example of accelerator-as-a-service in the cloud is SAP DASH [51], which presents the FPGA as a hardware data processing service to a cluster of databases, delegating the asynchronous execution of compute-intensive database operations.

4.2 Request Scheduling and Multi-tenancy

In the FaaS model, the FPGA host server is responsible for scheduling the requests to the FPGAs.

Each FPGA can instantiate either one large or multiple smaller SwiftSpatial kernels. Since an FPGA has fixed hardware resources, both methods are bounded by the same limit for resource consumption, thereby offering the same compute capabilities. When a single accelerator kernel is instantiated, user queries are queued and processed sequentially. Alternatively, multiple smaller kernels allow for the concurrent processing of several queries.

The difference between the two approaches lies in memory management and fairness. The multi-tenancy approach enhances fairness, preventing a situation where the accelerator is monopolized by a single query for an extended period, while other users are left waiting. However, this approach requires sharing memory between queries, which can limit the memory available for each query and potentially degrade performance compared to the single kernel instantiation.

5 Evaluation

We evaluate SwiftSpatial on real-world and synthesized datasets and compare it to a wide range of CPU- and GPU-based spatial join implementations. We show that SwiftSpatial outperforms the

most performant software baseline by up to 41.03× in terms of latency, while requiring 6.16× less power. The modular architecture of SwiftSpatial facilitates its deployment not only on data-center-grade FPGAs but also on embedded systems, showcasing its adaptability and potential for broad applicability.

5.1 Experimental Setup

Datasets. We evaluate SwitftSpatial on both real-world and synthesized datasets. For real-world datasets, we use the Open Street Maps (OSM) dataset made available by SpatialHadoop [25]. Specifically, we process its *buildings* subset for polygon generations (converted to MBRs) and use the *all nodes* subset for point objects. The dataset scales range from one hundred thousand to ten million objects, and we produce two different datasets for the spatial join operation. We also produce synthesized datasets of rectangles in uniform distribution. Specifically, we set the map size as 10K by 10K, within which each unit-square object is randomly and uniformly distributed.

Software. For CPU baselines, we evaluate both open-source software and our multi-threaded C++ synchronous traversal and PBSM implementations.

The open-source software baselines include PostGIS [8], Apache Sedona [1], and SpatialSpark [73], thus covering both spatial database and spatial big data frameworks. PostGIS is a spatial database that extends PostgreSQL with spatial functionalities. Apache Sedona, previously known as GeoSpark [75], is a system for processing large-scale spatial data. Even on a single server, Sedona carries out spatial joins on multiple-core CPUs by performing join on each data partition and subsequently merging the results. Similar to Sedona, SpatialSpark is an academic big spatial data framework developed on top of Apache Spark [77]. All the CPU baseline software above performs spatial join using R-trees. For each query, we perform a warmup run followed by three executions, from which we report the performance results. Since the datasets fit in main memory, the data already resides in memory during subsequent runs, avoiding data movement overhead from disks.

We implement multi-threaded C++ spatial join baselines, including R-tree synchronous traversal and PBSM. Our implementations are adapted from the single-threaded artifacts in [66]. For synchronous traversal, we implement two multi-threaded versions using OpenMP, covering both BFS and DFS. The first version applies a BFS traversal, identical to the FPGA implementation illustrated in §3.4. At each level of the BFS traversal, we consider the node pairs for joining as tasks to be assigned to various threads. Each thread is responsible for a subset of tasks, with a single thread subsequently merging the results. The second version combines DFS with BFS. The idea is to start the search with multi-threading BFS, and, once the number of node pairs to join surpasses the number of threads significantly, these tasks are dispatched to different threads. These threads manage the tasks using the conventional DFS synchronous traversal, after which a single thread merges the results. Specifically, when the task count is at least ten times greater than the hardware thread number, the algorithm switches to DFS. This is because assigning only one task per thread can lead to significant workload imbalances due to the varying DFS traversal time per task. For both BFS and BFS-DFS implementations, we evaluate two OpenMP scheduling strategies, static and dynamic, and report the best performance. The static schedule assigns the same number of tasks to each thread, while the dynamic schedule does not assume the number of tasks per thread and assigns a task to a thread as soon as it is idle. Our multi-threaded PBSM based on OpenMP supports arbitrary partition numbers and uses plane sweep to join the tiles. We adopt the one-dimensional PBSM, which partitions the data in one dimension and sweeps the data in the other dimension, because it has shown superior performance compared to the two-dimensional PBSM [69].

For GPU evaluations, we use cuSpatial [6], the only open-source GPU-based spatial join library we found. cuSpatial only supports Point-in-Polygon tests (points join polygons using the within predicate) rather than joining two polygon datasets. Consequently, we restrict our GPU evaluations

to these queries. Besides, cuSpatial employs quadtrees rather than R-trees as the index, because the boundaries of quadtree nodes can be computed simply with knowledge of the tree's levels and the node ID. However, this comes at the expense of an inferior indexing quality compared to R-trees [47]. The quadtree index is only constructed on the point dataset, and the Point-in-Polygon queries are performed through the use of polygon batches serving as window queries.

Hardware. The baseline CPU server contains an AMD EPYC 7313 16-core processor (7 nm) with a base frequency of 3 GHz and 256 GB of DDR4 memory. We run cuSpatial on an A100 SXM4 GPU (7nm) with 40 GB high-bandwidth memory (HBM). We evaluated SwiftSpatial on an AMD Alveo U250 FPGA (16 nm) equipped with 64 GB of DDR4 memory (4 x 16 GB). Both the CPU server and the FPGA have comparable costs on AWS [3, 5], at around 1.5 USD per hour on demand, while the GPU is more expensive, at around 4 USD per hour [4].

5.2 End-to-end Spatial Join Performance

We compare join latency between SwiftSpatial and the baseline CPU and GPU systems. We assume pre-constructed indexes on both datasets (R-trees and PBSM partitions), with associated costs reported separately in §5.9. For FPGA experiments, the join latency includes the join operation (kernel) and data movement (index and results) between the CPU and FPGA. For baselines, we measure only the join latency itself: the data and indexes are already loaded in memory (or GPU memory), and, for big data frameworks and PBSM, the data is already partitioned.

We report the best baseline and FPGA performance using the following configurations. For C++ multi-threaded synchronous traversal, we tune parameters, including node sizes, traversal strategies, and OpenMP scheduling policies. After evaluating different traversal and scheduling strategies, we find that a combination of BFS and dynamic scheduling provides the best performance in the majority of our experiments. For both SwiftSpatial and the C++ synchronous traversal, the maximum R-tree node size (the number of entries a node can accommodate) is set to 16, which is shown to be optimal in a later section (§5.3). For C++ multi-threaded PBSM, we evaluate different partition numbers ($10^2 \sim 10^5$) and partition-sweep directions (partition in one direction and sweep in the other) and report the best performance. For SwiftSpatial PBSM, we set the maximum tile size to accommodate up to 16 objects (based on the geometric mean of the two input datasets) as this setting shows optimal performance. With PostGIS, we configure the maximum parallel worker processes to 16, aligning with the available number of CPU cores on the server. For SpatialSpark, we assess varying numbers of partitions and find that the optimal setting is 64. In the case of cuSpatial, we fine-tune the average leaf node size in the quad-tree and set it as 128 for our experiments eventually. Since cuSpatial processes spatial join as window queries on the indexed point datasets, we use a batch size of 20K, such that the throughput is maximized without overflowing the GPU memory.

SwiftSpatial achieves significant speedup over all baselines. Figure 8 compares SwiftSpatial against our optimized C++ spatial join implementation, while Figure 9 compares it with existing spatial data processing systems. SwiftSpatial outperforms the multi-threaded synchronous traversal by 1.99~41.03×, the multi-threaded PBSM by 2.97~577.33×, the single-threaded C++ implementation by 19.37~395.93×, the single-threaded PBSM by 39.33~2428.92×, PostGIS by 188.24~9670.90×, Apache Sedona by 63.29~9225.98×, SpatialSpark by 1095.46~25626.81×, and cuSpatial by 186.60~21731.85×.

As shown in Figure 8, the PBSM version of SwiftSpatial shows the best performance across all datasets. PBSM allows SwiftSpatial to directly join tiles that are likely to yield results, unlike R-tree synchronous traversal, which involves multiple intermediate levels that require reading objects and writing intermediate results back.

For the CPU baselines, the multi-threaded synchronous traversal consistently delivers the best or near-optimal performance among the evaluated baseline systems. While PBSM showcases

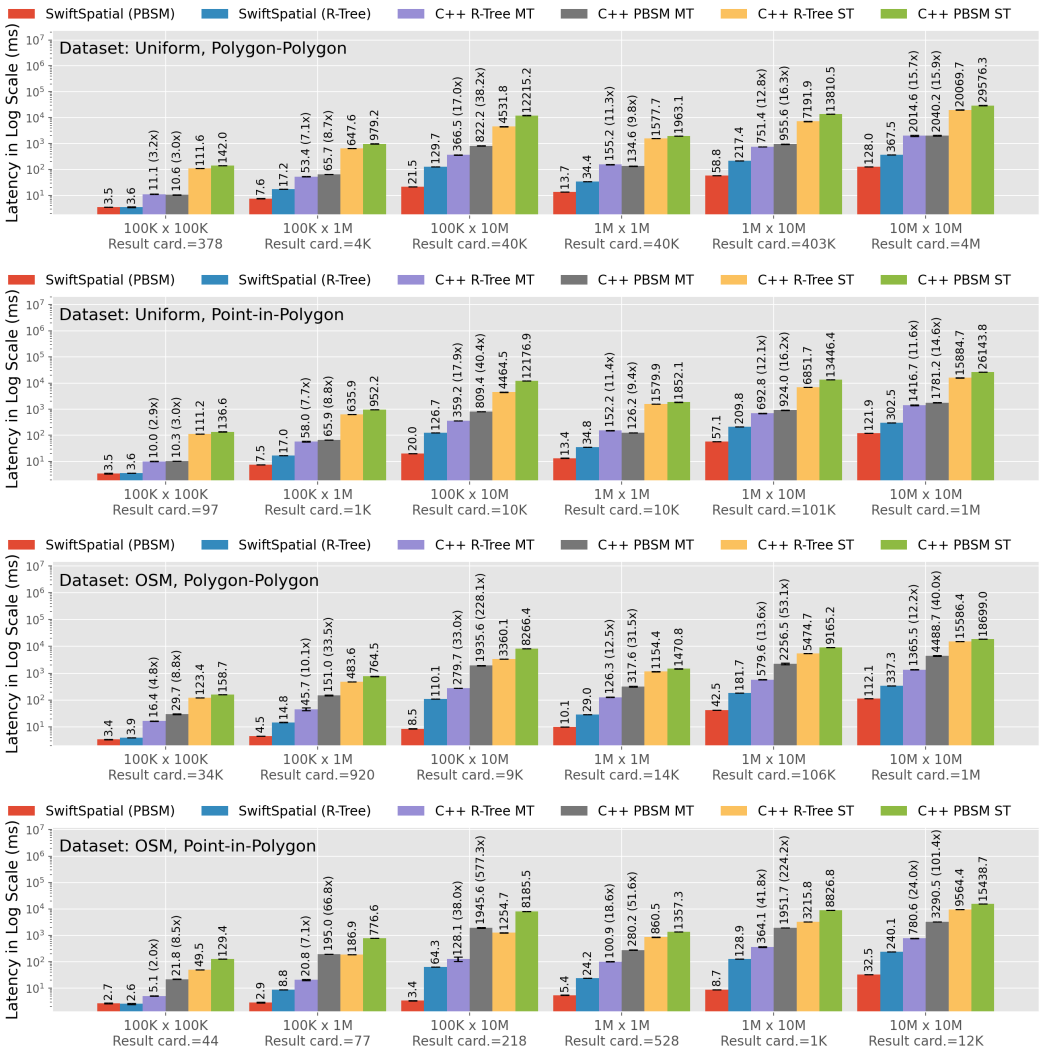


Fig. 8. Spatial join performance of SwiftSpatial compared to optimized CPU baselines across datasets, scales, and geometries.

near-linear scalability on Uniform datasets (14.5x latency reduction with 16 cores for 10Mx10M polygons), the single-threaded performance of PBSM lags behind that of synchronous traversal (same observation as in [66]), thus PBSM, in the best case, can only match the performance of synchronous traversal with 16 cores. However, PBSM's scalability depends on data distributions: the scalability falters on OSM datasets with a skewed object distribution, even with large partition numbers. Despite leveraging multi-core capabilities, PostGIS and the two big data frameworks tend to underperform single-threaded C++ synchronous traversal, as shown in Figure 8 and Figure 9. This underperformance might be attributed to the overhead of abstractions and, for the big data frameworks, inefficiencies related to Scala compared to C++ and the data shuffling costs.

CuSpatial, despite being GPU-based, proves to be one of the slowest baselines, even underperforming the single-thread C++ implementations. For the GPU performance measurement, we already indexed the datasets and copied the data to the GPU memory prior to measuring the join

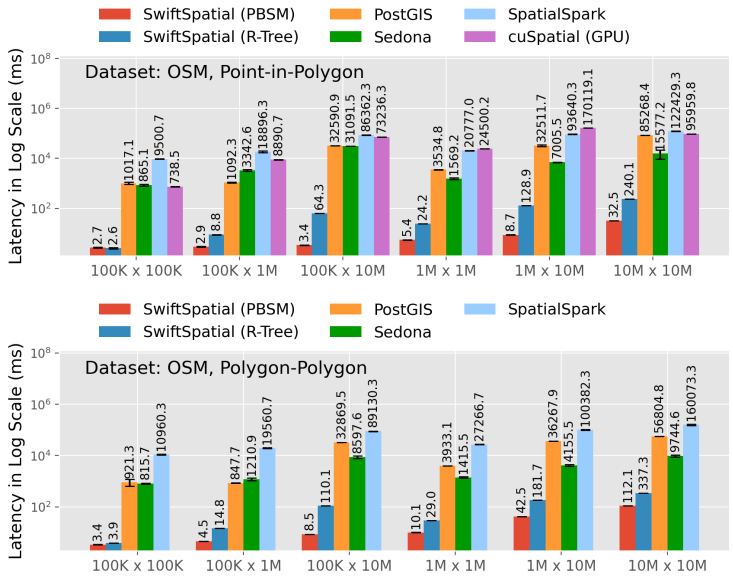


Fig. 9. Spatial join performance of SwiftSpatial versus CPU- and GPU-based spatial data systems.

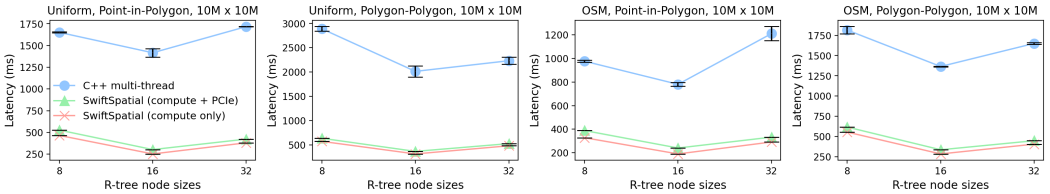


Fig. 10. The effect of different R-tree node sizes on spatial join performance given 16 threads or 16 join units.

performance. The low performance may stem from several factors. Firstly, the quadtree index implemented on the GPU is of lesser quality compared to R-trees [47]. Secondly, cuSpatial only supports indexing for one dataset since it only supports point indexing. Consequently, polygons are processed as batch queries, resulting in less efficiency compared to synchronous traversal with both datasets indexed. Third, without knowing the number of join results in advance, cuSpatial resorts to over-allocating memory, limiting query batch sizes. For example, even though none of the datasets evaluated produce more than 300 million result object pairs (well below the 40 GB of GPU memory), the maximum batch size that passed all tests is only 20K. This limitation results in sub-optimal GPU core utilization.

5.3 Effect of Node Sizes

We investigate the impact of varying R-tree node sizes, defined as the maximum number of objects or children a node can contain, on spatial join performance on CPU and FPGA. Typically, smaller node sizes can result in more efficient search space pruning during synchronous traversal, thereby reducing the total number of predicate evaluations. However, this comes at the cost of increased random memory accesses to read more small nodes.

Figure 10 shows the impact of node sizes on performance for both the 16-thread C++ synchronous traversal and the 16-join-unit accelerator. Both systems reach peak performance with a node size of 16. Despite smaller node sizes offering enhanced efficiency in pruning the search space, the increase in random memory accesses restricts the rate at which node pairs can be loaded into the

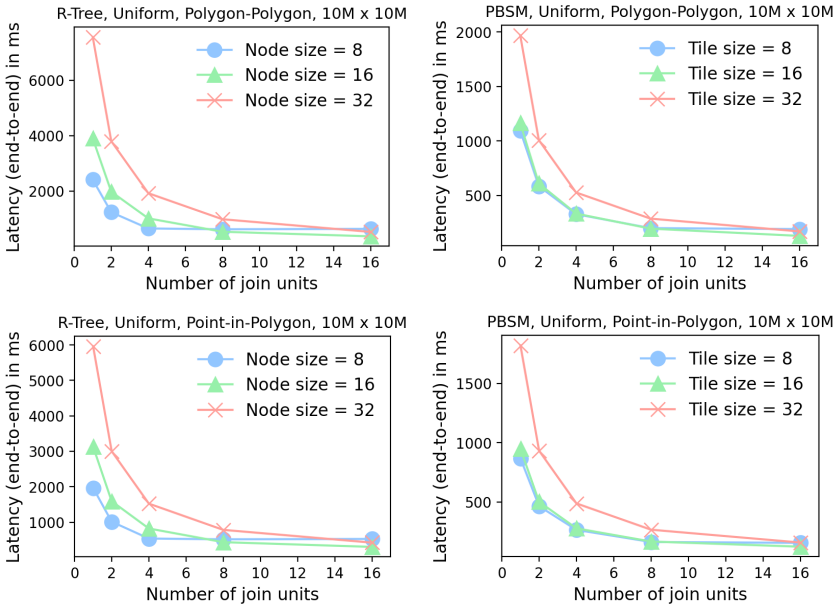


Fig. 11. The effect of R-tree node sizes or PBSM tile sizes when instantiating different numbers of join units.

processor or the accelerator. Conversely, with larger node sizes, the effectiveness of space pruning decreases, leading to more predicate evaluations between node pairs throughout the traversal.

5.4 Performance Scalability of the Join Units

We evaluate the performance scalability of the join units in both synchronous traversal and PBSM by instantiating one to sixteen units on SwiftSpatial. We first examine the impact of varying node sizes in relation to different numbers of join units. Subsequently, we evaluate the performance scalability across a range of node sizes in R-trees or tile sizes in PBSM.

The selection of optimal node sizes is related to the number of instantiated join units. Figure 11 compares the end-to-end join performance using different datasets and node or tile sizes. For synchronous traversal for the Polygon-Polygon join on the Uniform dataset (upperleft of Figure 11), the accelerator with a single join unit optimally favors a smaller node size of eight. As the quantity of join units escalates to eight, a node size of 16 emerges as the optimal selection. With an accelerator comprising 16 join units, a node size of 16 still retains its optimal status; however, the performance disparity between node sizes of 16 and 32 diminishes significantly compared to one join unit. The rationale behind this lies in the fact that, when a lesser amount of join units are instantiated, the computation resources dedicated to predicate evaluation are restricted, thereby making R-trees with smaller nodes or PBSM with smaller tiles more desirable choices due to their potential to reduce computation effectively. Contrarily, when there is a substantial number of join units instantiated, these units end up competing for memory access. Consequently, spatial join with smaller nodes becomes memory-bound, leading to a situation where some units are idle, waiting for data.

As node size increases, the performance scales better with the number of join units. Figure 12 shows the speedup observed when instantiating multiple join units. For synchronous traversal on the Uniform dataset (top left), the performance plateaus after instantiating four units when using a node size of eight, as the many short memory read operations cannot fully leverage the memory bandwidth. Conversely, with a node size of 32, the performance scales better by adding

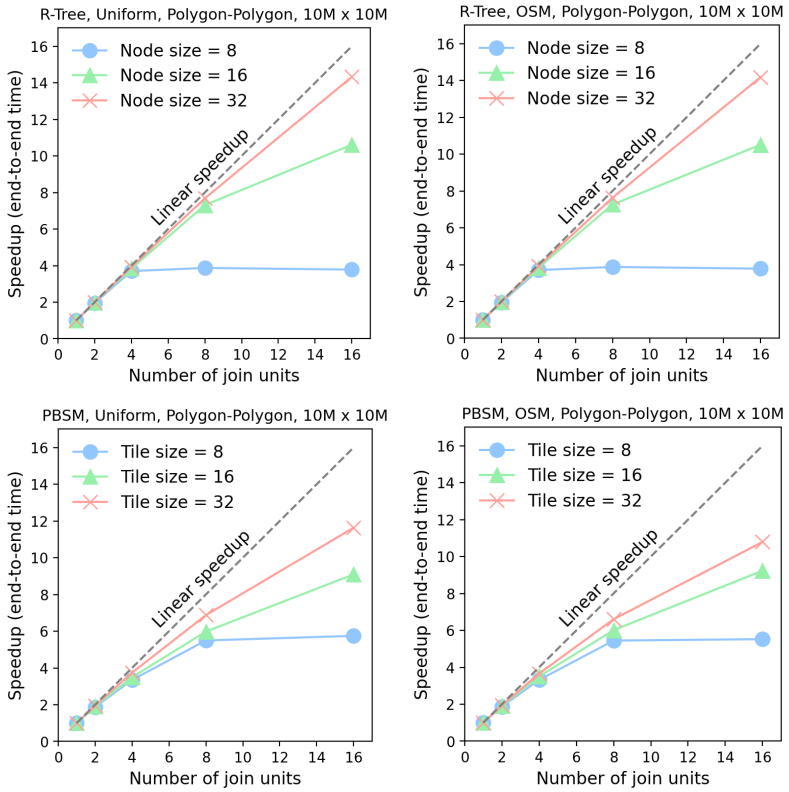


Fig. 12. The performance scalability of the join units using different R-tree node sizes or PBSM tile sizes.

more units, demonstrating an almost linear progression up to 16 join units. The same performance trend has been observed on the OSM dataset (top right). The PBSM implementation shows better performance scalability for small node sizes like eight (bottom of Figure 12), as it does not involve the intermediate result read and write operations in R-tree synchronous traversal.

5.5 How Good is a Hardware Join Unit?

In addition to the end-to-end performance evaluation, we microbenchmark the join units, specifically focusing on their efficiency in evaluating join predicates between nodes. For this purpose, we constructed an accelerator with a single join unit and fed R-tree node pairs of varying sizes into it.

The join unit nearly achieves the ideal performance rate of one predicate evaluation per cycle. Figure 13 shows the join unit performance given different node sizes. The plot on the left presents the number of clock cycles needed to perform a join between a pair of nodes, which includes the time needed for reading the data and evaluating the predicates. The plot on the right normalizes the performance as the average number of cycles needed per predicate evaluation. For instance, if both nodes contain 32 entries, then 32^2 predicates are evaluated during the join operation. If this operation takes 1066 cycles to complete, then the number of cycles per predicate evaluation is $1066/32^2 = 1.04$. As shown in the plot on the right, the performance of joining small nodes (with four or fewer entries) is constrained by the random DRAM accesses to fetch the input node pairs. Conversely, the performance is close to the optimal value (one predicate evaluation

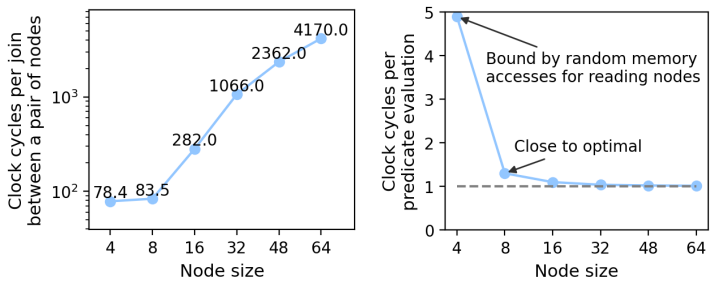


Fig. 13. Join unit can achieve near-optimal performance.

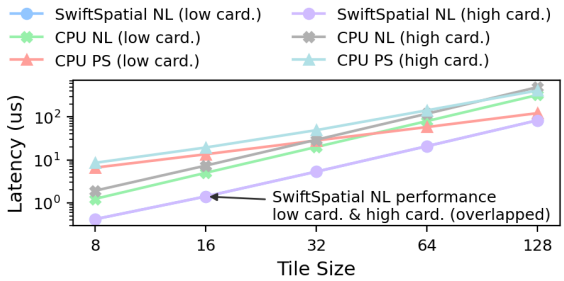


Fig. 14. Comparison of nested loop (NL) and plane sweep (PS) with different tile sizes and result cardinality.

per cycle) when a medium-sized node (eight or more entries) is employed. Here, the cycles per predicate range from 1.02~1.30 using node sizes of 8~64.

The SwiftSpatial join unit exhibits a significant advantage over software-based nested loop join or plane sweep. Figure 14 compares the latency of tile-level joins across varying tile sizes and result cardinalities, given a single SwiftSpatial join unit and a single-threaded C++ implementation. To modulate result cardinalities, we adjusted the edge lengths of the tiles and populated them with unit-length rectangles. As a reference, the low-cardinality configuration yields no results for tile sizes under 128, while the high-cardinality setup produces 2170 results when joining 128-object tiles. Several observations can be made from Figure 14. Firstly, even software nested loop join outperforms plane sweep for small and moderate tile sizes (less or equal to 32 and 64 for the low and high cardinality experiments, respectively). Secondly, the performance of plane sweep is more susceptible to data distributions, as the increasing number of objects in the sweep queues can substantially hinder its efficiency. Thirdly, the SwiftSpatial join unit performance is consistent regardless of result cardinalities, as shown by the overlapped performance curve in Figure 14. Lastly, the fast predicate evaluation facilitated by the hardware join units leads to a significant performance advantage over software nested loop and plane sweep up to a moderate tile size (128), which is around an order of magnitude greater than typical R-tree node sizes.

5.6 Hardware Resource Consumption

Table 1 shows the hardware resource usage of SwiftSpatial, configured with different numbers of join units in the kernel. The kernel stands for user-space logic, while the shell refers to the FPGA infrastructure, including the memory controller, the PCIe controller, etc., consuming a constant amount of resources. The resource categories include Lookup Tables (LUTs) used for combinational logic, Flip-Flops (FFs) functioning as registers, Block RAMs (BRAMs) serving as on-chip memory, and Digital Signal Processors (DSPs) responsible for floating-point operations.

Table 1. SwiftSpatial consumes few FPGA resources.

	LUT	FF	BRAM	DSP
Kernel (1 PE)	0.67%	0.44%	2.46%	0.16%
Kernel (2 PE)	0.87%	0.55%	3.65%	0.21%
Kernel (4 PE)	1.24%	0.75%	6.03%	0.34%
Kernel (8 PE)	1.96%	1.13%	10.79%	0.60%
Kernel (16 PE)	3.35%	1.60%	28.05%	1.12%
Shell	10.89%	9.21%	14.96%	0.11%
Shell + Kernel (16 PE)	14.24%	10.81%	43.01%	1.23%
FPGA Total	1,728,000	3,456,000	2,688	12,288

On a data center FPGA such as U250, an accelerator kernel equipped with 16 join units consumes less than 30% of the total hardware resources. Notably, the most substantial resource consumption is Block RAMs (BRAMs), accounting for 28.05% of the total resources, while other resources are far less utilized (below 4%). This is attributed to the fact that each join unit in SwiftSpatial requires minimal resources (LUT, FF, and DSP), while the FIFOs interconnecting the join units, burst buffers, and read and write units all demand BRAM slices. These FIFOs are used to buffer data within the processing pipeline, thereby minimizing stalls.

SwiftSpatial, when configured with a smaller number of join units, can be deployed on embedded FPGAs for edge spatial computing. For example, PYNQ Z2, one of the lowest-end CPU-FPGA SoC on the market, contains 106,400 FFs, 53,200 LUTs, 140 BRAMs, and 110 DSPs. Even under a conservative assumption that 60% of these resources can be allocated for user kernels, it would be feasible to host a SwiftSpatial kernel comprising one to two join units. By optimizing BRAM usage, such as employing shift registers in place of BRAMs to construct the FIFOs, it becomes possible to instantiate a SwiftSpatial kernel with up to four join units on the PYNQ Z2.

5.7 Power Consumption

We compare the power consumption between CPU, GPU, and FPGA for spatial join. For CPU, we measure the multi-threaded C++ implementation as it is the most performant software baseline; and we measure the GPU power consumption when running cuSpatial. In both cases, we executed the Point-Polygon join on the OSM 10M dataset continuously using a while loop and assessed the power using AMD RAPL (Running Average Power Limit) and NVIDIA System Management Interface, respectively. For the FPGA, we used Vivado to report the accelerator's power consumption.

Despite its superior performance, the power consumption of SwiftSpatial is only 23.48W, 6.16× lower than the CPU and 4.04× lower than the GPU. Thanks to the modest hardware resource utilization of SwiftSpatial (§5.6), the accelerator manages to sustain high performance while using minimal energy. The CPU power consumption (144.69W) is close to its Thermal Design Power (TDP) of 155W, indicating that the cores are fully utilized during the join operation. Contrarily, the GPU, despite having a 400W TDP, consumes only 95.01W during the join. This discrepancy likely stems from significant under-utilization of the many GPU cores: the substantial memory usage of cuSpatial limits the batch sizes during the join operation, thus restraining the core usage.

5.8 Filtering and Refinement

While this study primarily focuses on the filtering phase of spatial joins, we also evaluate the refinement performance to demonstrate the end-to-end benefits of SwiftSpatial in a spatial join pipeline.

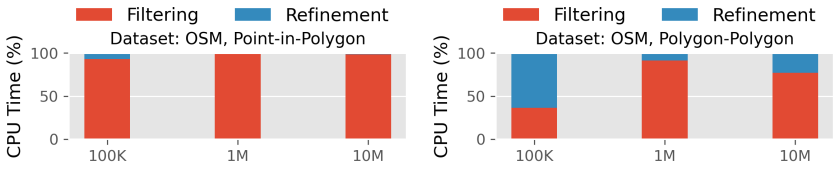


Fig. 15. Filtering versus refinement latency on CPUs.

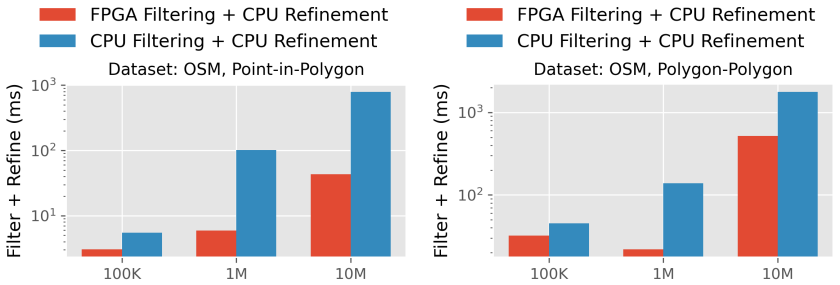


Fig. 16. End-to-end spatial join latency including filtering and refinement phases.

Figure 15 shows the percentage of time spent on the refinement stage in an end-to-end spatial join on the OSM dataset using CPUs. Overall, the filtering phase often consumes more CPU time than the refinement phase for this dataset. However, the time distribution between these phases is strongly influenced by output cardinality. For instance, a 10M polygon-polygon join produces 1M result pairs after filtering, whereas a 10M point-in-polygon join yields only 12K result pairs. Consequently, the polygon-polygon join spends approximately 23.0% of the total time on refinement, significantly more than 1.4% for the point-in-polygon case.

Figure 16 compares the performance of the end-to-end spatial join pipeline with and without SwiftSpatial, encompassing both filtering and refinement phases. In the case of SwiftSpatial, the filtered results are refined on a CPU server. Overall, SwiftSpatial accelerates the end-to-end spatial join pipeline by a factor of 1.4~18.3×, depending on the filtering speedup and the fraction of time spent on filtering within the spatial join pipeline.

5.9 Index Construction Cost

When interfaced with a spatial data management system, SwiftSpatial requires the R-trees to be constructed or the dataset to be partitioned just once by the CPU during the system's initialization phase. We study this one-time cost by implementing the R-tree index construction in C++ using the Sort-Tile-Recursive (STR) algorithm [48], which bulk loads the data to an R-tree using a recursive sorting and tiling strategy. We apply the parallel and vectorized sorting algorithm in the C++ Standard Template Library (STL). We also implemented a parallel data partitioning program: for CPU, it is a one-level partition, while for SwiftSpatial we add hierarchical partition to avoid tiles containing too many objects. Table 2 compares the time spent constructing indexes versus executing the join operation (using C++ multi-thread and SwiftSpatial) on several datasets of ten million objects, adopting a max node size or tile size of 16 as it demonstrates optimal join performance as evaluated in Section 5.3 and 5.4. Since the construction cost is typically higher than the join itself, spatial join based on R-trees would benefit more from iterative spatial joins [66], where at least one dataset is dynamic. In such scenarios, the R-tree is constructed once at the beginning, and subsequent insertions, updates, or deletions result in only minimal indexing overhead compared to

Table 2. Time consumption of R-tree index constructions and data partitioning on ten-million-object datasets.

	R-tree	Hierarchical Partition	Partition
Uniform Point-Polygon	9,212 ms	2,011 ms	508 ms
Uniform Polygon-Polygon	9,220 ms	2,051 ms	553 ms
OSM Point-Polygon	6,149 ms	2,047 ms	441 ms
OSM Polygon-Polygon	6,903 ms	2,662 ms	456 ms

bulk loading. For a one-off join, PBSM shows significantly less pre-processing overhead and thus would be a better choice.

6 Discussion

Reusability and extensions for alternative join algorithms. SwiftSpatial's modular design enables extensibility to other spatial join algorithms beyond PBSM and R-tree synchronous traversal. This versatility stems from the reusability of the accelerator's key component — the hardware join units — because joining small tiles is a major performance bottleneck in many spatial join algorithms [13, 52, 57, 69]. To support alternative spatial join algorithms, only the on-chip scheduler for tile-level join dispatching needs to be updated. For instance, to implement algorithms with complex control flow, such as synchronous traversal (§3.4), the scheduler should manage (a) reading upcoming tile-level join tasks from memory, (b) dispatching these tasks to the instantiated join units, and (c) terminating the join process once no further tasks remain.

Handling datasets larger than FPGA memory. So far, we have evaluated joins for which the indexed datasets and the join results fit in the FPGA memory. There are three potential solutions for larger-than-FPGA-memory joins. The first solution involves data partitioning, akin to big data frameworks. The data is partitioned, and the join operation is segmented into several sub-tasks, which are handled by multiple FPGAs before the results are aggregated. The second solution, given the significant speed advantage of FPGAs over software-based solutions, is that a single FPGA can process all data partitions iteratively. The third solution is to scale up the FPGA memory capacity: the FPGA, as a computational device on its own, can be equipped with an extended memory capacity, even up to a terabyte of memory per device [18].

Alternative spatial join algorithms on GPUs. While cuSpatial adopts quad-trees for spatial joins [6], and several academic efforts have implemented R-tree-based spatial joins [45, 50, 58, 72], PBSM presents another viable option due to its embarrassingly parallel nature. PBSM has the potential to outperform both quad-trees and R-trees on GPUs, as it can be efficiently parallelized across partitions on GPUs using multiple streaming multiprocessors.

However, we identify two performance challenges for GPU-based PBSM compared to SwiftSpatial. The first challenge is memory management. Without prior knowledge of the output cardinality, GPUs would likely require a two-pass join process: one pass to determine memory allocation per partition and a second pass to write the results. This approach is necessary because each streaming multiprocessor needs to determine the start and end addresses for its write operations. In contrast, SwiftSpatial avoids this overhead with a single processing pass. Results from all join units are streamed into a single write unit, which dynamically manages memory access using a self-incrementing counter. As a result, join units in SwiftSpatial do not need to handle memory write addresses. The second challenge is load balancing. Data skewness across PBSM partitions and the large number of streaming multiprocessors on GPUs can make it difficult to evenly distribute the workload. SwiftSpatial, however, mitigates load imbalance through its design: (a) it uses a smaller number of powerful join units, and (b) it supports both static and dynamic workload allocation

policies. The dynamic policy ensures efficient load distribution by assigning new tile pairs to join units only when they become available.

7 Related Work

To the best of our knowledge, SwiftSpatial is the first specialized hardware architecture for spatial join. Since we have introduced the landscape of spatial join research in §2, we now shift our focus to discussing related work in the realm of spatial data management techniques and hardware acceleration for data processing.

Indexes for spatial data. The R-tree [30], introduced by Guttman in 1984, has been and continues to be one of the most popular index structures for managing spatial data. Various efforts have been undertaken to enhance the topology of the R-tree, aiming to speed up spatial queries. For instance, both STR [48] and the Hilbert R-tree [41] improve R-tree topologies by more effective bulk loading strategies. On the other hand, the R^+ tree [62] and the R^* tree [11] focus on optimizing subtree selection for insertion and node splits, and recent advancements have incorporated reinforcement learning to guide those policies [28]. Apart from R-trees, there are other indexing techniques for spatial data. Quad-trees [59, 61] recursively divide space into four quadrants. KD-trees [54] split the space into two half-spaces along each of the k dimensions.

FPGAs in data centers. As Moore's Law has come to an end, substantial research has been devoted to processing data on hardware accelerators including FPGAs. FPGAs have demonstrated potential not only in relational joins [14] but also in other database operations [38] like regular expression matching queries [65], decision tree traversal [55], sketching [17, 44], and data partitioning [42]. Beyond these core database operations, FPGAs can accelerate graph traversal [15], high-dimensional approximate nearest neighbor search [37, 39, 40], data encryption and compression [16], embedding table lookups for recommender systems [35, 36], etc. Beyond developing accelerators on FPGAs, researchers have also explored infrastructure on FPGAs such as virtualization [43, 46, 78] and utilizing FPGA as part of the software system infrastructure [29, 31, 34, 70].

8 Conclusion

We introduce SwiftSpatial, a hardware accelerator architecture for spatial join. It consists of specialized join units featuring hybrid parallelism, dedicated memory management units, and an on-chip scheduler that coordinates these components. Prototyped on FPGA, SwiftSpatial achieves a latency improvement of up to 41.03× compared to the best-performing software baseline, all while requiring 6.16× less power supply. The modular structure of SwiftSpatial enables (a) its flexible instantiation on both data-center-grade FPGAs and embedded systems and (b) its adaptability to various spatial join algorithms, indicating its extensive applicability in future spatial data management systems.

Acknowledgments

We thank AMD for their generous donation of the Heterogeneous Accelerated Compute Clusters (HACC) at ETH Zurich (https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html), on which the experiments were conducted.

References

[1] [n. d.]. Apache Sedona. sedona.apache.org.
[2] [n. d.]. AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries. https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/.
[3] [n. d.]. AWS EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/
[4] [n. d.]. AWS EC2 P4 Instances. https://aws.amazon.com/ec2/instance-types/p4/
[5] [n. d.]. AWS F1 FPGA Instances. https://aws.amazon.com/ec2/instance-types/f1/
[6] [n. d.]. cuSpatial. https://github.com/rapidsai/cuspatial.

[7] [n. d.]. Intel FPGA SDK for OpenCL™ Software Technology. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.
[8] [n. d.]. PostGIS. http://postgis.net/.
[9] [n. d.]. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[10] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1009–1020.
[11] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.
[12] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1994. Multi-step processing of spatial joins. *Acm Sigmod Record* 23, 2 (1994), 197–208.
[13] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient processing of spatial joins using R-trees. *ACM SIGMOD Record* 22, 2 (1993), 237–246.
[14] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA useful for hash joins?. In *CIDR*.
[15] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.
[16] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware acceleration of compression and encryption in SAP HANA. In *48th International Conference on Very Large Databases (VLDB 2022)*.
[17] Monica Chiosa, Thomas B Preußer, and Gustavo Alonso. 2021. Skt: A one-pass multi-sketch data analytics accelerator. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2369–2382.
[18] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. 2022. Enzian: an open, general, CPU/FPGA platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 434–451.
[19] Mark De Berg, Marc Van Kreveld, Mark Overmars, Otfried Schwarzkopf, Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. *Computational Geometry: Introduction*. Springer.
[20] J-P Dittrich and Bernhard Seeger. 2000. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 535–546.
[21] Harish Doraiswamy and Juliana Freire. 2020. A gpu-friendly geometric data model and algebra for spatial queries. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1875–1885.
[22] Harish Doraiswamy and Juliana Freire. 2022. SPADE: GPU-Powered Spatial Database Engine for Commodity Hardware. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2669–2681.
[23] Ahmed Eldawy, Louai Alarabi, and Mohamed F Mokbel. 2015. Spatial partitioning techniques in SpatialHadoop. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1602–1605.
[24] Ahmed Eldawy and Mohamed F Mokbel. 2015. The era of big spatial data. In *2015 31st IEEE International Conference on Data Engineering Workshops*. IEEE, 42–49.
[25] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.
[26] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.
[27] Thanasis Georgiadis and Nikos Mamoulis. 2023. Raster Intervals: An Approximation Technique for Polygon Intersection Joins. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–18.
[28] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[29] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 417–433.
[30] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.
[31] Zhenhao He, Dario Korolija, Yu Zhu, Benjamin Ramhorst, Tristan Laan, Lucian Petrica, Michaela Blott, and Gustavo Alonso. 2023. ACCL+: an FPGA-Based Collective Engine for Distributed Applications. *arXiv preprint arXiv:2312.11742* (2023).

[32] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[33] Yun-Wu Huang, Ning Jing, Elke A Rundensteiner, et al. 1997. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *VLDB*, Vol. 97. 25–29.

[34] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1202–1213.

[35] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. 2021. MicroRec: efficient recommendation inference by hardware and data structure solutions. *Proceedings of Machine Learning and Systems* 3 (2021), 845–859.

[36] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. 2021. Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3097–3105.

[37] Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. 2024. Accelerating Graph-based Vector Search via Delayed-Synchronization Traversal. *arXiv preprint arXiv:2406.12385* (2024).

[38] Wenqi Jiang, Dario Korolija, and Gustavo Alonso. 2023. Data processing with fpgas on modern architectures. In *Companion of the 2023 International Conference on Management of Data*. 77–82.

[39] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, and Gustavo Alonso. 2023. Co-design Hardware and Algorithm for Vector Search. (2023), 1–15.

[40] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. 2025. Chameleon: a Heterogeneous and Disaggregated Accelerator System for Retrieval-Augmented Language Models. *Proceedings of the VLDB Endowment 18* (2025).

[41] Ibrahim Kamel and Christos Faloutsos. 1993. *Hilbert R-tree: An improved R-tree using fractals.* Technical Report.

[42] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. Fpga-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 433–445.

[43] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with {AmorphOS}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 107–127.

[44] Martin Kiefer, Ilias Poulakis, Eleni Tzirita Zacharatou, and Volker Markl. 2023. Optimistic Data Parallelism for FPGA-Accelerated Sketching. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1113–1125.

[45] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *J. Parallel and Distrib. Comput.* 73, 8 (2013), 1195–1207.

[46] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do {OS} abstractions make sense on {FPGAs}?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 991–1010.

[47] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. 2002. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 546–557.

[48] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings 13th international conference on data engineering*. IEEE, 497–506.

[49] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[50] Lijuan Luo, Martin DF Wong, and Lance Leong. 2012. Parallel implementation of R-trees on the GPU. In *17th Asia and South Pacific Design Automation Conference*. IEEE, 353–358.

[51] Norman May, Daniel Ritter, Andre Dossinger, Christian Färber, and Suleyman Demirsoy. 2023. DASH: Asynchronous Hardware Data Processing Services. In *13th Conference on Innovative Data Systems Research, CIDR*.

[52] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. 2013. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 701–712.

[53] Peter Ogden, David Thomas, and Peter Pietzuch. 2016. AT-GIS: highly parallel spatial query processing with associative transducers. In *Proceedings of the 2016 International Conference on Management of Data*. 1041–1054.

[54] Jack A Orenstein and Tim H Merrett. 1984. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 181–190.

[55] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. 2019. Lowering the latency of data processing pipelines through FPGA based hardware acceleration. *Proceedings of the VLDB Endowment* 13, 1 (2019), 71–85.

[56] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment* 11, 11 (2018), 1661–1673.
[57] Jignesh M Patel and David J DeWitt. 1996. Partition based spatial-merge join. *ACM Sigmod Record* 25, 2 (1996), 259–270.
[58] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 618–627.
[59] AndreasKipf HaraldLang VarunPandey RaulAlexandruPersa, Christoph Anneser Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter Boncz, and Thomas Neumann Alfons Kemper. 2020. Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins. (2020).
[60] Yeasir Rayhan and Walid G Aref. 2023. SIMD-ified R-tree Query Processing and Optimization. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*. 1–10.
[61] Hanan Samet and Robert E Webber. 1985. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics (TOG)* 4, 3 (1985), 182–222.
[62] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. (1987).
[63] Darius Šidlauskas and Christian S Jensen. 2014. Spatial joins in main memory: Implementation matters! *Proceedings of the VLDB Endowment* 8, 1 (2014), 97–100.
[64] Darius Šidlauskas, Simonas Šaltenis, Christian W Christiansen, Jan M Johansen, and Donatas Šaulys. 2009. Trees or grids? Indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL international conference on Advances in Geographic Information Systems*. 236–245.
[65] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 403–415.
[66] Benjamin Sowell, Marcos Vaz Salles, Tuan Cao, Alan Demers, and Johannes Gehrke. 2013. An experimental analysis of iterated spatial joins in main memory. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1882–1893.
[67] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2003. Hardware acceleration for spatial selections and joins. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 455–466.
[68] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. 2016. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment* 9, 13 (2016).
[69] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. Parallel in-memory evaluation of spatial joins. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 516–519.
[70] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. 2022. {FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 967–986.
[71] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 international conference on management of data*. 1071–1085.
[72] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on gpus using r-trees. In *Proceedings of the 2Nd ACM SIGSPATIAL international workshop on analytics for big geospatial data*. 23–31.
[73] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in cloud. In *2015 31st IEEE international conference on data engineering workshops*. IEEE, 34–41.
[74] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in cloud. In *2015 31st IEEE international conference on data engineering workshops*. IEEE, 34–41.
[75] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*. 1–4.
[76] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio 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.
[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.
[78] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 845–858.
[79] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store.. In *FAST*. 225–237.
[80] Geraldo Zimbrao and Jano Moreira de Souza. 1998. A raster approximation for processing of spatial joins. In *VLDB*, Vol. 98. 24–27.

Received October 2024; revised January 2025; accepted February 2025