

Article

Data-Driven Analysis for University Governance Modernization, Discipline Development, and Academic-Risk Management: A Spark-GPU Heterogeneous Acceleration Framework for Large-Scale Multi-Source Data Mining

Yong Huang

College of Education, Capital Normal University, Beijing 100048, China; 2230401076@cnu.edu.cn

Abstract

Large-scale spatial data mining contains two performance bottlenecks: iterative high-dimensional distance computation and irregular polygon verification. This study develops a Spark-compatible CPU-GPU acceleration framework for K-Means clustering and polygon spatial join. Spark is retained for data ingestion, partitioning, scheduling, and result reconstruction, whereas CUDA executes the dominant numerical and geometric kernels. SGK-Means combines flattened array communication, unified index mapping, K-Means++ initialization, Yinyang bound filtering, and a single Spark partition plus single-GPU strategy for the iterative clustering loop. SG-Join integrates CUDA refinement into Apache Sedona through Spark-side KD-Tree partitioning, GPU-side equal-grid indexing, MBR filtering, point-in-polygon (PIP) verification, edge-intersection (EI) verification, duplicate removal, and CUDA dynamic parallelism. Runtime is reported as end-to-end wall-clock time, including data conversion and host-device transfer. On the RTX6000 SGK-Means setting, the maximum speedup is 247.19 and the geometric-mean speedup is 6.21 across the sixteen dataset-k configurations; the Tesla A40 hardware comparison reports a maximum speedup of 270.15 on Higgs with $k = 10,000$. For SG-Join, the maximum speedup is 3.90 and the geometric-mean speedup is 1.99 across dataset and CPU-core settings. A configuration-level exact sign-test check is added for the retained paired runtime records, while the evaluation still distinguishes this directional check from repeated-run uncertainty analysis. These results indicate that Spark-GPU cooperation improves CPU-baseline runtime when the data layout, communication path, and CUDA kernels are matched to the dependency structure of each algorithm. The conclusion is restricted to the evaluated datasets, hardware settings, and baseline protocols. The revised evaluation treats output equivalence, run-to-run uncertainty, partial component sensitivity, and GPU-alternative comparison as explicit scope controls rather than as unsupported correctness or statistical-superiority claims.



Academic Editor: Jie Yang

Received: 7 May 2026

Revised: 24 June 2026

Accepted: 26 June 2026

Published: 30 June 2026

Copyright: © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: symmetry-aware computing; spatial data mining; Spark-GPU acceleration; heterogeneous computing; K-Means clustering; polygon spatial join; CUDA; Apache Sedona; Yinyang filtering; spatial partitioning

1. Introduction

The rapid deployment of mobile Internet services, 5G networks, sensing infrastructures, vehicle positioning systems, remote sensing platforms, and location-aware applications has changed geospatial data processing from a conventional database task into a

large-scale high-performance computing problem. Spatial objects are continuously produced as points, trajectories, raster blocks, building footprints, administrative polygons, road networks, and thematic layers. These data streams support urban planning, traffic analysis, surveying and mapping, energy management, land monitoring, and commercial location analytics. The technical requirement behind these applications is not only storage capacity but also low-latency extraction of spatial patterns from massive and heterogeneous datasets.

Traditional spatial analysis pipelines were designed for single-machine execution or for centralized CPU-oriented database engines. Such designs become inefficient when the number of objects, the dimensionality of attributes, or the geometric complexity of polygons increases. The major bottlenecks include limited memory bandwidth, high latency in iterative computation, insufficient throughput for pairwise geometric predicates, and weak scalability under uneven spatial distributions. In spatial clustering, each iteration requires repeated distance computation between a large number of samples and cluster centroids. In polygon spatial join, each object from one layer may need to be tested against a large number of objects from another layer, and every candidate pair may further require exact geometric verification.

Distributed big-data frameworks such as Apache Spark 3.5.1 reduce the limitations of a single machine by partitioning data and scheduling tasks across multiple executors. However, pure CPU-based distributed execution does not always deliver proportional speedup for spatial data mining. Iterative algorithms introduce synchronization and shuffle overheads. Polygon operations introduce irregular workloads because different polygons contain different numbers of vertices and occupy different spatial extents. As a result, CPU parallelism alone may be unable to convert data-level parallelism into sufficient runtime reduction.

Heterogeneous computing provides a direct engineering path for this problem [1,2]. CPU cores are effective for control-dominant operations such as data loading, partitioning, task scheduling, and result aggregation, whereas GPUs provide massive fine-grained parallelism and high memory bandwidth for regular numerical kernels. A Spark-GPU platform can therefore map scheduling-intensive tasks to Spark and computation-intensive kernels to CUDA. This mapping is asymmetric at the hardware level but symmetry-aware at the algorithmic level: homogeneous data points, centroid comparisons, grid cells, and polygon-pair candidates are transformed into balanced and repeatable parallel tasks whenever the data structure permits.

The key difficulty is that adding GPUs to Spark does not automatically accelerate spatial data mining. Data must cross the Java/C/CUDA boundary, host-device transfer must be controlled, and GPU kernels must respect memory hierarchy and thread hierarchy [3,4]. An inefficient design can spend more time in serialization, format conversion, and inter-device communication than it saves through parallel computation. Therefore, the design target is a low-intrusion Spark-GPU execution path that preserves Spark-side data processing while moving only the computationally dominant kernels to CUDA.

High-performance geospatial platforms have mainly evolved along distributed storage, distributed indexing, and cloud-based spatial analytics. SpatialHadoop extends Hadoop with spatial data types, spatial partitioning, spatial indexes, and spatial operators [5]. GeoSpark, later developed as Apache Sedona, extends Spark and provides spatial RDDs, spatial SQL, spatial partitioners, and distributed spatial queries [6,7]. Cloud platforms such as Google Earth Engine, Microsoft Azure geospatial services, and Sentinel Hub provide scalable access to remote sensing and geospatial data-processing services [8,9]. These systems improve scalability for large datasets, but their core execution engines still rely primarily on CPU-side distributed scheduling and CPU-side geometric processing [10].

Heterogeneous big-data engines attempt to integrate GPUs into data-parallel frameworks. HeteroSpark introduces a plug-in Spark-GPU acceleration mechanism for machine learning workloads [11]. Spark-GPU extends Spark to support CPU-GPU hybrid analytics and addresses data transfer, data layout, and scheduling [12]. GFlink integrates GPU resources into a Flink-based in-memory computing architecture and introduces JVM-GPU communication strategies, GPU caching, and adaptive scheduling [13]. RAPIDS provides CUDA-accelerated primitives, GPU-enabled dataframes, and acceleration support for selected Spark operations [14].

These platforms show that GPUs can accelerate big-data workloads, but generic acceleration mechanisms are insufficient for complex spatial algorithms. Spatial mining algorithms have specific data dependencies, memory-access patterns, pruning opportunities, and correctness constraints. K-Means has simple data-level parallelism but severe iterative synchronization pressure under distributed execution. Polygon spatial join has strong pruning potential but requires exact verification after coarse filtering. Consequently, spatial algorithms still require customized CUDA kernels, compact data layouts, and algorithm-aware Spark-GPU data paths rather than direct reuse of a general GPU scheduler.

K-Means clustering has been parallelized using MapReduce, MPI, FPGA, and GPU architectures [15,16]. MapReduce-based approaches distribute sample assignment and centroid update across mappers and reducers, improving scalability for large datasets but adding shuffle overhead in every iteration [17]. MPI-based approaches reduce communication by using message passing and collective operations, but the programmer must explicitly manage synchronization and load distribution [18]. FPGA implementations exploit hardware pipelines and low power consumption, but their flexibility is limited when the data dimension, cluster number, or distance metric changes [19]. GPU implementations accelerate the distance-computation phase by mapping samples or sample-centroid pairs to many threads, but they must control memory transfer and centroid-update reductions [20,21].

The Spark MLlib implementation of K-Means uses RDD partitioning, broadcast variables, and map-reduce aggregation to compute assignments and update centroids [22]. This design is convenient and robust for large datasets, but the iterative structure requires repeated coordination among partitions. When the number of clusters is large, the assignment phase becomes the dominant computation; when the dataset is distributed, each iteration may also introduce communication overhead. Therefore, the central issue is not whether K-Means can be parallelized, but where its iteration loop should be executed to maximize computation throughput while minimizing synchronization.

Spatial join has also been studied under CPU-distributed and GPU-parallel settings. CPU-distributed systems such as SpatialHadoop, GeoSpark, SpatialSpark, Simba, and GeoMesa partition data and build spatial indexes to reduce unnecessary comparisons [23]. GPU-based methods accelerate point-in-polygon tests, edge intersection, spatial indexing, and filtering-refinement pipelines [24,25]. Nevertheless, single-GPU spatial join is constrained by device memory, and pure CPU-distributed polygon spatial join can suffer from high latency when object counts and polygon complexity grow. Polygon-to-polygon joins are especially expensive because the candidate generation phase and the refinement phase are both data-dependent [26].

Similar scalable data-processing demands can also be found in heterogeneous scheduling, distributed coordination, climate and engineering modeling, smart manufacturing, online platforms, computer vision, IoT systems, secure localization, sparse reconstruction, scientific machine learning, and data-driven optimization [27–79]. In financial risk management, Zhang et al. developed AD-VGF, a generative-adversarial-network-based model for credit-card risk identification in the digital economy, further demonstrating the need

for scalable feature learning and reliable data-driven decision support [80]. These studies further show that modern multi-source data mining requires efficient feature processing, scalable execution, and reliable system deployment.

The main parallelization gaps are therefore algorithm-specific. For K-Means, distributed Spark execution may increase shuffle and synchronization cost during iterative convergence checking. For polygon spatial join, single-GPU or pure CPU-distributed execution may encounter memory limits, workload imbalance, spatial skew, and costly exact geometry verification. A practical Spark-GPU spatial mining framework must handle both algorithms with different mapping strategies rather than applying a single acceleration template.

A Spark-GPU platform should assign scheduling-intensive tasks to Spark and computation-intensive kernels to CUDA. Spark handles data ingestion, partitioning, and distributed scheduling, while CUDA executes arithmetic-intensive or geometry-intensive kernels with regular thread-level independence. The closest related GPU systems, including RAPIDS/cuML, GCMF, and SPADE, demonstrate the feasibility of GPU acceleration for machine learning and spatial operators, but their execution models differ from the low-intrusion Spark/JNI/CUDA integration studied here. Accordingly, this paper does not frame the reported CPU-baseline speedups as superiority over all GPU systems. Instead, it evaluates whether two Spark-compatible operators can reduce wall-clock runtime relative to the corresponding Spark MLib and GeoSpark/Sedona workflows under matched data-processing pipelines. SGK-Means targets high- k clustering workloads in which sample-centroid distance evaluation dominates; SG-Join targets polygon join workloads in which candidate filtering, exact geometric verification, and partition-level load imbalance dominate.

The key contributions are as follows. First, the paper gives an operational Spark-GPU mapping principle in which task-level symmetry refers to homogeneous work units executed by the same kernel template, while hardware asymmetry refers to assigning control-intensive operations to CPUs and throughput-intensive kernels to GPUs. Second, SGK-Means is implemented as a low-communication Spark-compatible GPU operator using one-dimensional arrays, unified indexing, K-Means++ initialization, Yinyang filtering, and GPU-local convergence checking. Third, SG-Join is implemented as a Spark/Sedona-compatible polygon join operator using Spark-side KD-Tree partitioning, GPU-side equal-grid indexing, MBR pruning, PIP/EI refinement, duplicate removal, and dynamic parallelism. Fourth, the evaluation reports the acceleration conditions and the limitations of the framework, including cluster number, dataset size, GPU device, partition number, CPU-core setting, spatial dataset characteristics, speedup aggregation, and single-GPU memory constraints.

2. System Model and Heterogeneous Computing Preliminaries

2.1. Parallel Performance Metrics and Optimization Objectives

The execution target is a heterogeneous data-processing pipeline in which Spark performs host-side data operations and CUDA performs device-side computation. Runtime is the primary performance variable because both evaluated algorithms are time-consuming kernels in spatial data mining workflows. The most direct metric is speedup, which compares the execution time of a baseline method with the execution time of the accelerated method. Formula (1) defines the speedup index (S) and the derived runtime-change index (R_T) used in the revised tables. The baseline runtime is denoted by T_{baseline} , and the accelerated runtime is denoted by $T_{\text{accelerated}}$. A positive value of R_T indicates runtime reduction relative to the CPU baseline.

$$S = T_{\text{baseline}}/T_{\text{accelerated}}; \quad R_T = (1 - 1/S) \times 100\% \quad (1)$$

Parallel efficiency evaluates the utilization of parallel resources when p processing units are used. Formula (2) normalizes the speedup by the number of processors or parallel execution units and gives the equivalent runtime-ratio expression. Although GPU execution uses many more lightweight threads than CPU execution, the metric remains useful for understanding why more partitions or more CPU cores do not always lead to proportional acceleration.

$$E_p = S/p = T_{\text{baseline}}/(p \times T_{\text{accelerated}}) \quad (2)$$

Scalability is interpreted as the ability of the algorithm to maintain or improve performance when the dataset size, cluster number, polygon count, or partition count increases. Amdahl-type reasoning is also relevant: acceleration is bounded by the portion of the execution path that remains serial or communication-dominated. In a Spark-GPU setting, the serial and communication components include Spark scheduling, Java object preparation, JNI conversion, host-device transfer, and result reconstruction. Therefore, the optimization objective is not only to accelerate arithmetic kernels but also to reduce data movement and avoid unnecessary synchronization.

Because the recorded experiments are end-to-end workflow timings, the current evaluation does not decompose each run into separate serial, transfer, and GPU-kernel fractions. Therefore, Amdahl-type reasoning is used here as a design constraint for reducing communication and synchronization overhead, rather than as a fitted quantitative performance model.

The two algorithms have different optimization priorities. SGK-Means focuses on reducing iterative communication and redundant distance computation. SG-Join focuses on reducing candidate pairs, balancing spatial workloads, and accelerating exact geometric verification. Both algorithms are evaluated using runtime and speedup, while the discussion also considers partition sensitivity, CPU-core sensitivity, GPU-device sensitivity, and dataset-feature sensitivity.

2.2. Spark Execution Model and CPU-Side Task Scheduling

Spark represents distributed data through resilient distributed datasets and executes programs through a DAG-based scheduling mechanism. An RDD is split into partitions, and each partition is processed by tasks running inside executors. A user action triggers a job, the DAG scheduler divides the job into stages according to dependency boundaries, and the task scheduler assigns task sets to available executors. Narrow dependencies can be executed locally, whereas wide dependencies introduce shuffle and require data movement across executors.

Spark maintains the logical dataflow and controls task scheduling, while GPU kernels are invoked only inside selected computational stages. This avoids rewriting the whole data-processing pipeline and keeps compatibility with existing Spark/Sedona preprocessing. For SGK-Means, Spark is deliberately prevented from distributing the iterative centroid-update loop across many workers, because repeated cross-partition synchronization would reduce the benefit of GPU acceleration. For SG-Join, Spark partitioning is retained because spatial partitions can be processed independently before the final duplicate removal.

Partition number is a critical engineering parameter. Too few partitions may underutilize available executors and GPUs, whereas too many partitions can increase scheduling, serialization, and data-transfer overhead. Shuffle cost is especially important because both algorithms contain operations that can generate global dependencies. The proposed designs therefore choose different Spark-side strategies according to algorithm structure: compact single-node GPU execution for SGK-Means and distributed partition-level GPU execution for SG-Join.

2.3. CUDA Memory and Thread Hierarchies for GPU Kernels

CUDA execution separates the host and the device. The host CPU launches kernels, transfers data, and collects results. The GPU device executes kernels through streaming multiprocessors, each of which schedules warps composed of 32 threads. The memory hierarchy includes off-chip global memory, constant memory, texture/surface memory, local memory, and on-chip registers and shared memory. Kernel performance strongly depends on memory locality, coalesced access, occupancy, and the ratio of arithmetic operations to memory transactions.

The CUDA memory model is directly relevant to both algorithms because data prepared in Spark memory must be converted into device arrays, copied to global memory, and then accessed by many threads. Shared memory is used only for data that must be reused by threads within a block, such as local accumulators or temporary intersection counts. Global memory stores the main flattened data arrays because the datasets are large and must be visible to all kernels.

Table 1 now reports the actual memory-placement decisions used in the Spark-GPU data path rather than a generic CUDA memory taxonomy. The table links each algorithmic state variable to its preferred storage location, access pattern, implementation rationale, and the overhead or correctness risk controlled by that design choice.

Table 1. Memory-placement decisions used in the Spark-GPU data path.

Algorithmic State	Preferred Device-Side Storage	Dominant Access Pattern	Reason for This Placement	Risk Controlled
Flattened sample matrix X	Global memory	Coalesced sequential reads by sample-owned threads	The matrix is large and persists across assignment iterations; contiguous arrays avoid nested Java-object traversal after JNI transfer.	Serialization overhead and non-coalesced object access
Centroid array C	Global memory, with block-local reuse when applicable	Repeated reads during sample-centroid distance evaluation	Centroids are much smaller than X but must remain visible to all assignment threads after each update.	Redundant host-device transfer during the iterative loop
Assignment labels P	Global memory	One write per sample and one read during convergence checking	Only labels are returned to Spark after convergence, which keeps the output compact.	Unnecessary vector-level result transfer
Bound arrays U and L	Global memory	Per-sample read/write during filtering	The Yinyang upper and lower bounds must persist across iterations and are updated independently by sample threads.	CPU-side scanning of pruning states
Polygon vertices and MBR arrays	Global memory	Partition-local indexed reads by candidate-pair threads	Variable-size polygon data are flattened into arrays so that GPU refinement can operate without nested spatial objects.	Irregular JVM object traversal and excessive candidate-transfer cost
Intersection counters and temporary flags	Shared memory and registers	Block-local accumulation and predicate flags	Temporary values are small and reused only within one verification block.	Global-memory contention during PIP/EI refinement

CUDA kernels are organized as grids, blocks, and threads. A grid contains multiple blocks, and a block contains multiple threads. Threads within the same block can cooperate through shared memory and synchronization, whereas blocks are scheduled independently. This hierarchy is used in SGK-Means by assigning independent data points to threads. It is used in SG-Join by assigning candidate polygon pairs and grid partitions to CUDA blocks and threads.

The CUDA thread hierarchy described above also explains why workload balance is necessary. A warp executes most efficiently when its threads follow similar control paths. Spatial join violates this condition because different polygon pairs may require different numbers of edge tests or point-in-polygon operations. SG-Join therefore introduces CUDA dynamic parallelism to launch child grids for irregular verification tasks. SGK-Means has a more regular structure, and its main optimization is to reduce unnecessary centroid comparisons before they reach the distance-computation kernel.

2.4. Unified Spark-GPU Data Path and Symmetry-Aware Mapping Principle

The common execution path uses Spark for data ingestion and preprocessing, JNI for Java/C/CUDA interaction, and CUDA kernels for compute-intensive operations. Data are loaded and preprocessed in Spark, transformed into primitive arrays before JNI transfer, processed by GPU kernels, and returned as compact results for subsequent Spark-side analysis. The design does not replace Spark; it inserts GPU computation into selected stages with limited changes to the original processing flow.

For SGK-Means, Spark prepares the samples and receives only cluster identifiers after GPU execution. For SG-Join, Spark partitions spatial layers, sends partition-local polygon arrays to the GPU, and receives matching polygon-pair identifiers. The JNI boundary is therefore crossed using compact arrays rather than nested spatial objects or Spark RDD structures.

In this paper, symmetry is used as an operational computing concept rather than as a geometric property. A set of work units $W = \{w_1, w_2, \dots, w_m\}$ is treated as task-symmetric when the same kernel template can process each unit and when the unit has only local data dependencies after partitioning. CPU-GPU asymmetry is then used deliberately: Spark executes irregular control, partition construction, scheduling, and aggregation, whereas CUDA executes the repeated numerical or geometric kernels. This definition directly drives two different mappings. SGK-Means localizes the globally synchronized iteration loop to one GPU because centroid updates create cross-partition dependencies, while SG-Join distributes independent spatial partitions through Spark before GPU-side refinement. Thus, the symmetry-aware design criterion is the preservation of uniform parallel work units under asymmetric heterogeneous resources.

3. SGK-Means: Spark-GPU Parallel K-Means Clustering

3.1. Problem Formulation and Parallel Bottleneck of K-Means

Given a dataset $X = \{x_1, x_2, \dots, x_n\}$ with n samples and d dimensions, K-Means partitions the samples into k clusters. The centroid set is denoted as $C = \{c_1, c_2, \dots, c_k\}$. In each iteration, every sample is assigned to its nearest centroid, and each centroid is updated using the mean of the samples assigned to that cluster. The dominant computation is the sample-centroid distance evaluation, whose direct complexity is $O(k \times n \times d)$ per iteration.

The Spark MLlib baseline parallelizes the assignment and update phases using partition-level map-reduce operations. Current centroids are broadcast to partitions, samples are assigned locally, and partial sums are reduced to update the centroid set. This strategy is robust, but every iteration depends on global centroid states. When k grows, local distance computation becomes expensive; when data are distributed, the aggrega-

tion and convergence check introduce synchronization cost. The bottleneck is therefore a combination of arithmetic intensity and iterative communication.

SGK-Means uses K-Means++ to improve initial centroid selection and uses the Yinyang upper–lower-bound strategy to avoid unnecessary distance calculations. For a sample x , $b(x)$ denotes the current best centroid, $u(x)$ denotes the upper bound to $b(x)$, and $\ell(x, G_r)$ denotes the lower bound from x to centroid group G_r . The bound structure is valid only when the same distance function and numerical precision are used in the assignment and bound-update steps. Formula (3) defines the basic relationship used before group and local filtering are applied.

$$u(x) \geq d(x, b(x)); \quad \ell(x, G_r) \leq \min_{c \in G_r, c \neq b(x)} d(x, c) \quad (3)$$

Formula (3) is the basis of the filtering logic. If the lower bound remains sufficiently larger than the updated upper bound after centroid movement, the sample cannot change its cluster assignment and its distance computation can be skipped. This pruning is highly suitable for GPU execution because each sample can be checked independently and assigned to a CUDA thread.

3.2. Data Flattening and JNI-Based Communication Reduction

Spark stores samples as RDD partitions containing vector objects. Directly transferring these objects to CUDA would require object serialization, structural traversal, and type conversion. SGK-Means replaces object-level transmission with primitive one-dimensional arrays. Each sample vector is flattened according to a unified row-major index. The value of dimension j in sample i is placed at a deterministic offset, enabling both Spark and CUDA to address the same data without reconstructing object hierarchies.

The original RDD vectors are converted into one JNI array before transfer. This conversion improves communication efficiency because JNI transfers primitive arrays more efficiently than nested Java objects. It also improves GPU memory access because the device reads contiguous numeric arrays instead of irregular object fields.

The unified indexing scheme further reduces result-transfer volume. The GPU does not need to return updated samples or full vectors. It only returns the assigned centroid index for each sample. Consequently, the return-transfer space is reported as a vector-to-label reduction ratio.

$$C_{\text{vec}} = O(nd), \quad C_{\text{label}} = O(n), \quad C_{\text{label}}/C_{\text{vec}} = O(1/d) \quad (4)$$

The reduction shown above is critical for K-Means because the algorithm may process millions of samples and dozens of dimensions. For small values of k , the GPU computation may be short, and communication can dominate. For large values of k , arithmetic cost grows rapidly, and the same compact data path allows the GPU to expose its parallel advantage without excessive transfer overhead.

3.3. Single Spark Partition Plus Single-GPU Execution Strategy

Although K-Means contains multiple data-parallel steps, distributing the full iterative loop across Spark nodes is not always efficient. After each iteration, all partitions must contribute to centroid update and convergence checking. A multi-node Spark implementation therefore introduces shuffle and synchronization in every iteration. A multi-node Spark-GPU implementation would additionally require communication between Spark executors and GPUs, and possibly indirect synchronization among GPUs through the Spark driver.

SGK-Means executes the clustering loop inside one Spark partition connected to one GPU. Spark prepares the flattened input and transfers it to the GPU through JNI. The GPU performs centroid initialization support, point assignment, bound maintenance, centroid update, and convergence checking. After convergence, only the compact cluster labels are returned to Spark.

The design is based on a practical observation: for K-Means, excessive distribution can be harmful when communication cost exceeds the benefit of additional parallelism. A single GPU can provide enough thread-level parallelism for the dominant distance computations, while Spark remains responsible for input preparation and downstream processing. This strategy is especially effective when k is large and the assignment phase dominates runtime.

This strategy does not mean that the whole framework is a non-distributed system. The term large-scale in this paper refers to the scale of the input data and to a Spark-compatible heterogeneous data-processing workflow. SGK-Means is an algorithm-specific Spark-GPU operator: Spark is used for input preparation, workflow integration, and result handling, whereas the iterative assignment-update loop is deliberately localized to one GPU to avoid repeated global centroid synchronization. SG-Join uses a different mapping because its spatial partitions are independent after Spark-side partitioning and can therefore be distributed across Spark tasks before GPU-side refinement. Thus, the framework is heterogeneous and Spark-based, but the degree of distribution is selected according to the dependency structure of each algorithm.

3.4. CUDA Kernel Design with Yinyang Filtering

The CUDA implementation contains three kernel-level functions: point initialization with bound construction, point reassignment with Yinyang filtering, and convergence checking. K-Means++ is used to choose initial centroids before GPU-local iteration. The centroid set is divided into t groups, where $t = \lceil k/10 \rceil$ following the grouping strategy of Yinyang K-Means, and group-level lower bounds are maintained for pruning. Each sample is processed independently by CUDA threads, making the assignment stage highly parallel. The stopping rule combines assignment stability with the maximum-iteration bound used in the paired Spark MLlib configuration.

The global filtering condition is given in Formula (5). If the updated lower bound $\ell^+(x)$ is not smaller than the updated upper bound $u^+(x)$, the sample remains assigned to its current centroid. The expression uses centroid movement δ to update the bounds between two consecutive iterations.

$$u^+(x) = u(x) + \delta_{b(x)}, \quad \ell^+(x) = \ell(x) - \delta_{\max}; \quad \text{keep } b(x) \text{ if } \ell^+(x) \geq u^+(x) \quad (5)$$

Formula (6) extends global filtering to centroid groups. For a centroid group G_i , an updated group lower bound $\ell^+(x, G_i)$ is maintained. If the group bound cannot beat the current upper bound, all centroids in that group are skipped. This avoids a large number of centroid comparisons when k is high.

$$\ell^+(x, G_i) = \ell(x, G_i) - \delta_i, \quad \delta_i = \max_{c \in G_i} \delta_c; \quad \text{skip } G_i \text{ if } \ell^+(x, G_i) \geq u^+(x) \quad (6)$$

Formula (7) gives the local filtering condition in a compact decision form. Exact distance evaluation is performed only for candidates in surviving groups; otherwise, the candidate distance $d(x, c)$ is skipped. This condition is applied only to the groups that survive the group filter.

$$\text{evaluate } c \in G_i \text{ only when } \ell^+(x, G_i) < u^+(x); \text{ otherwise skip } d(x, c) \quad (7)$$

Formula (8) updates centroid c_j by reusing the previous assignment information. V_j and V_j' denote the sample sets assigned to cluster j before and after reassignment. Instead of recomputing the centroid from all samples, the update removes samples that leave the cluster and adds samples that enter the cluster.

$$c_j' = (|V_j| c_j - \sum_{x \in V_j \setminus V_j'} x + \sum_{x \in V_j' \setminus V_j} x) / |V_j'| \quad (8)$$

Table 2 reorganizes the SGK-Means workflow as a host-device responsibility map rather than as line-by-line pseudocode. The table emphasizes which state remains on Spark, which computation is executed on CUDA, and which consistency checkpoint is used before the next stage.

Table 2. A host-device responsibility map of the SGK-Means workflow.

Workflow Stage	Spark-Side Responsibility	CUDA-Side Responsibility	State Passed Forward	Consistency Checkpoint
Input compaction	Load RDD vectors and convert them into primitive arrays.	Receive contiguous arrays through JNI.	Flattened X and initial parameter set	The same n , d , k , distance metric, and numerical precision are used by the baseline and accelerated paths.
Centroid initialization	Provide matched input data and clustering configuration.	Support K-Means++ centroid selection and centroid-group construction.	Initial C and group set G	Initialization settings are kept consistent for runtime comparison; shared initial centroids are required for output-equivalence checking.
Bound construction	No repeated Spark shuffle is triggered.	Initialize labels, upper bounds U, and group lower bounds L.	P, U, and L	Each sample owns one assignment state and one bound state.
Filtered reassignment	Keep the iterative loop outside distributed Spark synchronization.	Apply global, group, and local Yinyang filtering before exact distance evaluation.	Updated P, U, and L	Exact distances are evaluated only when the bound conditions cannot exclude a centroid or group.
Centroid update	Avoid cross-partition aggregation inside the loop.	Update centroids using assignment changes and GPU-local reductions.	Updated C	The update follows the same Lloyd-type mean rule under the matched distance metric.
Termination and return	Receive compact labels and reconstruct downstream results.	Check assignment stability and stop under the matched iteration limit.	Final label vector P	Only labels, not full vectors, cross back to Spark after convergence.

Table 3 summarizes the initialization and bound-construction safeguards used before filtered reassignment. The presentation focuses on the generated state variables and their roles in later pruning, instead of repeating the implementation sequence as kernel pseudocode.

Table 4 presents the reassignment procedure as a filtering decision hierarchy. Each row identifies the bound condition, the action taken when the condition is satisfied, and the correctness boundary under which distance calculations can be safely skipped.

Table 3. Initialization-state construction and pruning safeguards in SGK-Means.

Initialization Objective	Implementation Choice	Generated State	Role in Later Computation
Create deterministic sample ownership	Map each sample index to a CUDA thread through the flattened row-major offset.	Thread-local sample identifier	Enables independent bound construction without reconstructing Spark vector objects.
Select initial assignment	Compare the sample with the selected centroid set under the same distance function.	Initial label in P	Provides the starting cluster for Lloyd-type iteration.
Set the first upper bound	Store the distance from the sample to its assigned centroid.	Upper-bound value U	Allows later iterations to decide whether the current assignment can remain unchanged.
Build group lower bounds	Record the closest known distance information for centroid groups other than the assigned group.	Lower-bound vector L by group	Provides the group-level pruning condition before local centroid checks.
Preserve numerical comparability	Use the same precision specification for distance, centroid, and bound arrays.	Comparable CPU/GPU state variables	Limits differences caused by data-layout conversion rather than algorithmic logic.
Prepare convergence tracking	Store the current assignment state before the next reassignment stage.	Previous and current label states	Supports GPU-local convergence checking without a CPU-side full scan.

Table 4. Yinyang filtering decision layers used during SGK-Means reassignment.

Filtering Layer	Condition Examined	Action When the Condition Excludes a Candidate	Computation Avoided	Correctness Boundary
Current-center tightening	Whether the relaxed upper bound remains reliable after centroid movement	Recompute only the distance to the currently assigned centroid when tightening is needed.	Full centroid scan before validating the current bound	Uses the same distance metric as the assignment step.
Global group test	Whether the smallest updated group lower bound is still no smaller than the updated upper bound	Keep the current assignment.	All candidate centroid distances for that sample	Valid only after upper and lower bounds have been updated by centroid movements.
Group-level pruning	Whether a specific group lower bound can beat the current upper bound	Skip the entire centroid group.	All centroid comparisons inside that group	The group bound must be conservative after subtracting the maximum movement in the group.
Local centroid filtering	Whether an individual surviving centroid can improve the current assignment	Skip exact distance for the centroid when the local bound is sufficient.	One exact distance evaluation	Applied only to groups that survive the group-level test.
No-prune fallback	Bound conditions cannot exclude a group or centroid	Perform exact distance evaluation and update P, U, and L when needed.	None; exact calculation is retained.	Preserves the Lloyd-type assignment decision when pruning is inconclusive.

Table 5 separates convergence control from label-return control. This makes clear that the GPU-local convergence check only determines whether another assignment-update iteration is required, whereas Spark receives compact labels after the iteration loop has terminated.

Table 5. Convergence and result-return control in SGK-Means.

Control Aspect	Implementation in SGK-Means	Data Moved Across Spark-GPU Boundary	Reason for the Design
Assignment-state comparison	Previous and current labels are compared on the GPU after each reassignment stage.	No per-iteration label vector is returned to Spark.	Avoids CPU-side scanning and repeated Spark synchronization.
Convergence flag	A block/thread update marks the global convergence flag false when any label changes.	Only the final convergence state affects loop continuation.	A single flag is cheaper than transferring all assignment states each iteration.
Iteration bound	The GPU-local loop also respects the matched maximum-iteration setting.	No additional Spark action is needed during the loop.	Keeps the accelerated path comparable with the paired Spark MLlib setting.
Result reconstruction	After termination, CUDA returns only the final cluster label of each sample.	Label vector P rather than the full sample matrix X	Reduces return traffic from vector-level output to label-level output.
Evidence boundary	The table describes execution control and data movement, not a stored SSE comparison.	No unrecorded SSE values are introduced.	Prevents the manuscript from overstating output-equivalence evidence beyond the retained records.

Tables 2–5 jointly define the SGK-Means execution contract. Spark is responsible for input preparation and downstream use of the labels, while CUDA maintains assignment, bound, centroid, and convergence states during the iterative loop. This revised presentation avoids reproducing implementation pseudocode and instead clarifies the computational purpose, data movement, and correctness condition of each stage.

3.5. Complexity and Implementation Summary of SGK-Means

The direct assignment complexity of K-Means is $O(k \times n \times d)$ per iteration, where n is the number of samples, d is the feature dimension, and k is the number of centroids. SGK-Means does not change this worst-case order, because the Yinyang bounds may fail to prune any centroid when centroid movements are large or clusters overlap strongly. In the practical filtered case, the assignment cost can be described as $O(n \times t + n \times m \times d)$, where t is the number of centroid groups and m is the average number of centroids that survive local filtering for one sample. The acceleration therefore comes from reducing the effective number of exact distance evaluations and from executing the remaining evaluations on CUDA threads, not from changing the theoretical worst-case complexity class. In typical filtered iterations, m is expected to be much smaller than k , although this relation is data-dependent and is not assumed in the worst-case bound.

The communication complexity is reduced by flattening input data and returning only cluster identifiers. The input is transferred once as contiguous arrays, and the output is returned as a label vector. This design is well matched to the single Spark partition plus single-GPU strategy because no inter-GPU synchronization is required during the iteration loop. It also prevents repeated Spark shuffle during convergence checking.

The memory-access design uses global memory for flattened samples, centroid arrays, assignment labels, and bound arrays. Threads access sample values through deterministic

offsets, and each sample-thread follows the same assignment-filtering sequence. The implementation therefore combines task partitioning, communication reduction, and execution-strategy optimization. It is most beneficial when k , n , or d creates sufficient arithmetic intensity to amortize JNI and host-device transfer costs.

4. SG-Join: Spark-GPU Parallel Polygon Spatial Join

4.1. Polygon Spatial-Join Definition and Computational Bottleneck

Given two spatial object sets A and B , polygon spatial join returns all object pairs that satisfy a spatial predicate $\text{rel}(a, b)$. Formula (9) gives the same join definition in cross-product notation. The predicate can represent intersection, containment, touching, or another topological relation required by the query.

$$J_{\text{rel}}(A, B) = \{(a, b) \in A \times B : \text{rel}(a, b) = \text{true}\} \quad (9)$$

Polygon-to-polygon spatial join is computationally intensive because it combines pairwise candidate generation with exact geometric verification. A naive join compares many objects from one layer with many objects from the other layer. Even after spatial indexing, candidate pairs must be verified using geometric operations over polygon vertices and edges. The cost grows rapidly when polygon counts are high, polygon shapes are complex, or spatial overlap is dense.

Distributed CPU systems such as GeoSpark/Apache Sedona, SpatialHadoop, and Simba reduce candidate counts through spatial partitioning and indexing. Nevertheless, CPU-only execution remains limited by the throughput of geometric predicates and by skew among partitions. GPU acceleration can improve the refinement phase, but a single GPU cannot directly process arbitrarily large distributed datasets, and irregular polygon complexity can create load imbalance within GPU kernels. SG-Join addresses both issues by combining Spark-side distributed partitioning with GPU-side filter-refinement kernels.

4.2. Spark-Side Partitioning and Low-Intrusion Sedona Reconstruction

SG-Join is embedded into the Apache Sedona processing flow. Spark reads spatial layers, performs preprocessing, constructs the first-level spatial partitioning, and invokes GPU kernels through JNI. The original spatial-join operator is reconstructed at the Java level so that the GPU module can be called without changing the upper-level data-processing interface.

Spark prepares candidate data and transfers arrays through the JNI interface. The CUDA library performs MBR filtering, edge-intersection tests, and point-in-polygon tests. The result is returned as spatial-join pairs. This architecture preserves low intrusion: upstream loading and preprocessing and downstream result analysis remain compatible with Spark/Sedona.

When a polygon overlaps multiple index cells or partitions, copies are created in the corresponding partitions. Duplicate join results may therefore be generated, but they are removed after the join. This strategy trades limited storage redundancy for complete candidate coverage and avoids missing valid spatial relationships.

Several spatial index types can be used for partitioning. Quadtree partitioning can adapt to dense regions but may create imbalanced tree depth. R-Tree partitioning creates balanced bounding structures but may require additional handling for uncovered objects. KD-Tree partitioning provides better workload balance for the evaluated distributed Spark setting and is therefore selected as the Spark-side partitioner of SG-Join.

The experimental comparison uses the KD-Tree partitioning, equal-grid indexing, and Sedona baseline settings described in the method and reproducibility tables.

4.3. JNI Development Path and Distributed GPU Data Flow

The JNI development path connects Java-based Spark execution with C/CUDA kernels. A Java class defines the native spatial-join method and its primitive-array parameters. The Java bytecode is compiled, and a C header file is generated from the native method signature. The CUDA source file implements the generated method, includes the JNI header, and is compiled into a dynamic library. The Java code then loads the library and invokes the native method like a normal Java function.

The programming chain is Java plus JNI plus C plus CUDA. This chain is necessary because Spark and Sedona are executed in the JVM, whereas CUDA kernels are implemented in C/CUDA. SG-Join uses primitive arrays at this boundary to avoid transmitting nested polygon objects.

Spatial RDDs are partitioned by Spark, and each executor processes its assigned partition. Each executor communicates with the GPU device on the same physical node. This creates two parallel levels before CUDA dynamic parallelism is considered: distributed executor-level parallelism across Spark nodes and multiprocessor-level parallelism inside each GPU.

Spatial join partitions can be processed independently after the first-level spatial partitioning. No global centroid state must be synchronized after each iteration. Only the final join-pair union and duplicate removal require global aggregation. Therefore, SG-Join keeps distributed Spark scheduling while offloading partition-local geometric computation to GPUs.

4.4. GPU-Side Two-Stage Filter-Refinement Model

The GPU-side SG-Join kernel is organized as equal-grid indexing, MBR filtering, exact verification, and result deduplication. Equal-grid indexing creates a second-level partition inside the GPU to improve locality. MBR filtering removes polygon pairs whose bounding rectangles cannot satisfy the spatial predicate. Candidate pairs that pass MBR filtering are refined using exact geometry tests.

The same grid cell is mapped to a CUDA block or a group of blocks, which improves sequential memory access for partition-local polygon arrays. The MBR filter is the first pruning stage. It is fast because it uses only rectangle coordinates, but it is not exact because polygon interiors and edges may occupy only part of the rectangle.

The MBR stage can produce candidate pairs even when the true polygons do not intersect. It can also produce containment-like MBR relations even when the true polygon relation is intersection or disjoint. Therefore, PIPTest and EITest are both required. PIPTest verifies containment by testing whether a point of one polygon lies inside the other polygon. EITest verifies intersection by testing whether polygon edges intersect inside the common MBR.

The common minimum bounding rectangle is denoted as $CMBR = MBR_i \cap MBR_j$. If $CMBR$ equals one of the two $MBRs$, containment is possible and PIPTest is invoked. If $CMBR$ is non-empty but not equal to either MBR , edge intersection is the primary verification operation. When PIPTest returns false, EITest is invoked as a compensation step to preserve correctness, because an MBR containment relation can still correspond to polygon intersection rather than polygon containment.

Table 6 reformulates the SG-Join filter-refinement stage as a verification decision matrix. The table distinguishes MBR-generated candidates from final join results and specifies which exact predicate is used for each spatial relationship case.

Table 7 summarizes the design choices of the point-in-polygon verification stage. The table highlights the role of ray construction, block-level counting, parity reduction, and the compensation path to EITest when containment is not confirmed.

Table 6. Verification decision matrix for GPU-side SG-Join refinement.

MBR Relationship After Equal-Grid Indexing	Exact Verification Path	Final Flag Rule	Duplicate-Control Role	Correctness Note
No overlap between MBR _i and MBR _j	No PIP/EI refinement is launched.	The pair is discarded.	No result is generated.	A true polygon relation cannot exist when the two bounding rectangles are disjoint.
MBR _i contains MBR _j	Run PIPTest for the containment candidate; run EITest if PIPTest is false.	The pair is retained only if PIPTest or EITest is true.	Boundary copies are still deduplicated later.	MBR containment is only a candidate relation, so exact verification is required.
MBR _j contains MBR _i	Run PIPTest in the opposite containment direction; run EITest if needed.	The pair is retained only after exact verification.	Duplicate pairs are removed using pair identifiers.	The opposite direction prevents missing containment candidates.
MBRs overlap but neither contains the other	Run EITest within the common MBR region.	The pair is retained only when an edge intersection is detected.	Only true verification flags enter the final union.	Edge intersection must occur inside the common MBR if it exists.
Pair appears in multiple cells or partitions	Do not re-evaluate semantic truth; apply final pair-level deduplication.	One unique pair remains in R.	Removes duplicates caused by spatial boundary replication.	Completeness is prioritized during partitioning, and uniqueness is restored after refinement.

Table 7. Design choices and safeguards in CUDA point-in-polygon verification.

Design Element	CUDA Realization	Purpose	Limitation or Follow-Up Control
Test point selection	Use a selected vertex from the candidate polygon associated with the containment case.	Convert containment verification into a point-in-polygon decision.	A false result does not imply disjointness, so EITest is used as compensation.
Ray construction	Construct a ray from the selected point and test it against polygon edges.	Implement the odd-even rule for containment.	Degenerate edge cases are handled by the same predicate convention used in the verification routine.
Edge workload distribution	Assign edge checks to threads within the block.	Exploit parallelism over polygon edges.	Large or irregular polygons may still create uneven work, which motivates dynamic parallelism.
Shared counting state	Store per-thread intersection counts in block-local shared memory.	Reduce global-memory traffic during parity computation.	The counter is temporary and is not part of the final result set.
Parity decision	A block-level reduction produces the odd/even intersection count.	Set the verification flag for containment candidates.	Pairs with false PIPTest are passed to EITest when intersection remains possible.

Table 8 summarizes edge-intersection verification from the viewpoint of pruning, shared-memory reuse, and result-flag update. All CUDA-based verification tests were conducted on a workstation equipped with an NVIDIA GeForce RTX 4090 GPU (NVIDIA Corporation, Santa Clara, CA, USA), using CUDA Toolkit 12.2. The same hardware and software environment was used for all edge-pruning and verification procedures. It complements Table 7 by covering polygon pairs that require explicit edge-level intersection checking.

Table 8. Edge-intersection verification design and pruning logic.

Verification Operation	CUDA Realization	Pruning Basis	Output Effect
Common-region construction	Compute CMBR as the intersection of the two polygon MBRs.	Only edges crossing or inside CMBR can contribute to an intersection.	Restricts the edge tests to the geometrically relevant region.
Base-edge reuse	Load the selected edge from one polygon into shared memory within the verification block.	The same base edge can be compared against multiple candidate edges.	Reduces repeated global-memory reads.
Candidate-edge screening	Threads scan assigned edges from the other polygon and first test their relevance to CMBR.	Edges outside the common region are discarded before exact intersection.	Lowens the number of exact segment-intersection predicates.
Exact segment test	Apply the edge-intersection predicate only to surviving edge pairs.	The predicate is used after MBR and CMBR pruning.	Marks the polygon pair as true when any intersecting edge pair is found.
Result-flag update	Write the positive verification state to the pair flag F.	Only verified pairs are collected after kernel execution.	Separates candidate generation from the final spatial-join output.

The correctness of the filter-refinement model follows two geometric facts. First, if polygon P_1 contains polygon P_2 , then MBR_1 contains MBR_2 ; otherwise, the minimum bounding rectangle of P_2 would not be minimal. Second, if an edge from P_1 intersects an edge from P_2 , then the intersection point must be inside CMBR. Therefore, PIPTest and EITest together are sufficient to validate containment and intersection candidates produced by MBR filtering.

The final SG-Join output is defined as the set R of polygon pairs whose verification flag is true after PIPTest, EITest, and duplicate removal. Candidate generation by MBR filtering does not determine the final join result; it only reduces the number of pairs entering exact verification. This distinction clarifies that runtime acceleration is evaluated for the same spatial predicate and the same input layers as the GeoSpark/Sedona baseline.

4.5. CUDA Dynamic Parallelism for Load Imbalance Mitigation

Polygon spatial join has irregular computation. Different grid cells contain different numbers of polygons, and different polygons contain different numbers of vertices. PIPTest and EITest also have different computational costs. PIPTest may need to scan many edges to count ray intersections, whereas EITest may prune many edges through CMBR. If all candidate pairs are forced into a static kernel structure, some threads finish quickly while others execute long geometry checks, causing warp divergence and workload imbalance.

CUDA dynamic parallelism allows a parent grid running on the GPU to launch child grids without returning control to the CPU. The parent grid performs coarse candidate

processing and invokes child grids only when a candidate pair requires expensive exact verification. This avoids repeated CPU-GPU communication and allows the GPU to adjust the parallel granularity according to actual polygon complexity.

Dense grid cells can launch more child work, while simple candidate pairs terminate after MBR filtering. PIPTest and EITest are launched only when required by the MBR relation and the verification state. This creates a dynamic mapping from irregular polygon-pair workloads to GPU execution resources and improves load balance without changing the Spark-side partitioning interface.

5. Experimental Evaluation and Discussion

5.1. Experimental Setup and Baselines

The SGK-Means experiments compare the proposed Spark-GPU implementation with Spark MLlib K-Means. The Spark MLlib runtime is denoted by T_{baseline} , and the SGK-Means runtime is denoted by $T_{\text{accelerated}}$; these values are used to compute S and R_T . The experiments use four public datasets from the UCI Machine Learning Repository with sample counts from 0.52 million to 11 million and dimensions from 18 to 90. The cluster number k is varied to test the influence of clustering scale.

Table 9 reports the SGK-Means experimental environment. Two GPU environments are used to evaluate hardware generality. The first uses an NVIDIA Quadro RTX6000 with 24 GB memory (NVIDIA Corporation, Santa Clara, CA, USA), and the second uses an NVIDIA Tesla A40 with 48 GB memory (NVIDIA Corporation, Santa Clara, CA, USA). The software stack is consistent across the environments.

Table 9. SGK-Means runtime environment and device settings.

Configuration Group	Item	Specification
Group 01 hardware	Processor	Intel(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz
Group 01 hardware	Host memory	128 GB
Group 01 hardware	GPU device	NVIDIA Quadro RTX6000
Group 01 hardware	Device memory	24 GB
Group 02 hardware	Processor	Intel(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz
Group 02 hardware	Host memory	256 GB
Group 02 hardware	GPU device	NVIDIA Tesla A40
Group 02 hardware	Device memory	48 GB
Software	OS	CentOS 7.9
Software	CUDA	CUDA 11.7
Software	Spark	Spark 3.0.0
Software	Scala	Scala 2.12.15

The SG-Join experiments compare the proposed method with the GeoSpark/ Apache Sedona spatial-join implementation. The baseline uses KDB-Tree as the global index and R-Tree as the local index, which is a common setting for efficient Sedona spatial join. The experiments run on a distributed Spark environment with two physical nodes. Each node has the same hardware and software configuration, as shown in Table 10.

Table 10. SG-Join distributed runtime environment.

Configuration Group	Item	Specification
Hardware	Processor	Intel(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz
Hardware	Host memory	256 GB
Hardware	GPU device	NVIDIA Tesla A40
Hardware	Device memory	48 GB
Software	OS	CentOS 7.9
Software	CUDA	CUDA 11.7
Software	Spark	Spark 3.0.0
Software	Scala	Scala 2.12.15
Software	Sedona	Sedona 3.0_2.12

Table 11 lists the spatial datasets used in SG-Join. Each group contains two polygon layers used as the inputs of one spatial-join query. Group A is small, Group B is medium-sized with hundreds of thousands of polygons, and Group C is large with millions of polygons. These groups test different polygon counts, file sizes, and spatial distributions.

Table 11. Spatial dataset parameters.

Group	Dataset Name	Number of Polygons	File Size
A	Global_State_Province	8457	48 MB
A	Global_Urban_Area	12,154	42 MB
B	USA_block_boundary	242,575	1.9 GB
B	USA_Water_Bodies	456,151	850 MB
C	Yunnan_jsPolygon	1,561,656	1.1 GB
C	Yunnan_Rooftop	4,489,165	1.2 GB

The baselines and parameter settings are selected to evaluate two different acceleration mechanisms. SGK-Means is evaluated under variations in dataset size, dimension, cluster number, and GPU device. SG-Join is evaluated under variations in partition number, CPU core count, and dataset characteristics. All speedups are computed from end-to-end wall-clock runtimes rather than isolated kernel timings. The measured interval includes algorithm-specific data preparation, JNI conversion, host-device transfer, CUDA execution, Spark-side aggregation, duplicate removal when applicable, and result reconstruction. The retained experimental records contain the reported aggregate runtimes, but they do not contain five independent repetitions for every configuration. Therefore, the revised manuscript reports observed wall-clock comparisons and aggregate speedup summaries, while avoiding unsupported standard deviations, confidence intervals, Friedman/Wilcoxon tests, or statistical-superiority claims based on run-to-run distributions. To strengthen the runtime comparison with the retained paired records, an exact two-sided sign test is added in Section 5.6. The test counts how many paired configurations have speedup $S > 1$ under the null assumption that the accelerated and baseline workflows are equally likely to be faster across paired configurations. This check is used only as a configuration-level directional test; it does not estimate runtime variance and does not replace repeated-run means, standard deviations, or confidence intervals. The reproducibility protocol and paired-comparison settings are summarized in Table 12.

Table 12. Reproducibility protocol and paired-comparison settings.

Item	Configuration or Clarification
SGK-Means baseline	Spark MLlib K-Means with the same input dataset and cluster number k as the accelerated implementation.
SGK-Means initialization	K-Means++ initialization is used before GPU-local iteration; shared initial centroids are required when output equivalence rather than runtime comparison is evaluated.
Stopping rule	Assignment stability is checked on the GPU; the paired Spark MLlib setting uses $\text{maxIter} = 20$ and $\text{tol} = 1.0 \times 10^{-4}$.
Centroid grouping	The number of centroid groups is $t = \lceil k/10 \rceil$, consistent with the Yinyang grouping strategy.
Floating-point precision	Distance, centroid, and bound arrays are specified as double-precision numeric arrays to align the CPU and GPU comparison path.
CUDA thread mapping	SGK-Means maps one sample to one CUDA thread in the assignment-related kernels; SG-Join maps candidate polygon pairs and verification subtasks to CUDA threads.
PIPTest block-level storage	PIPTest uses the shared array <code>cutLineNum</code> so the block-level intersection count follows the 256-entry shared-memory design described in Table 7.
SG-Join baseline	GeoSpark/Apache Sedona spatial join with KDB-Tree global partitioning and R-Tree local indexing.
SG-Join variables	Partition counts of 500, 1000, 1500, and 2000; executor-core settings from 4×1 to 4×8 ; dataset groups A-C.
Code and data materials	Public datasets, environment settings, parameter tables, and implementation availability are clarified in the Data and Code Availability statements.

5.2. SGK-Means Performance Across Datasets and Clustering Scales

Table 13 reports the runtime comparison between Spark MLlib K-Means and SGK-Means on four datasets. The dataset size n , dimension d , and cluster number k are varied to expose the computational behavior of the algorithm. The results show that SGK-Means accelerates K-Means for most settings and that the speedup increases sharply when k becomes large.

For the runtime tables, the column S/R_T (%) reports speedup together with the derived runtime-change percentage from Formula (1). Positive values of R_T indicate runtime reduction, whereas negative values indicate slowdown relative to the CPU baseline.

The strongest acceleration in Table 13 appears on the Higgs dataset when $k = 10,000$. Spark MLlib requires 152,724.16 s, whereas SGK-Means requires 617.84 s, giving a speedup of 247.19 on the RTX6000 setting. A similar pattern appears on SuSy, where the speedup reaches 220.47 at $k = 10,000$. The geometric-mean speedup across the sixteen Table 13 configurations is 6.21, while the arithmetic mean is 36.88 and is strongly affected by the largest $k = 10,000$ cases. These settings have high assignment complexity because every

sample must be compared with many centroids in the baseline. SGK-Means benefits from both CUDA parallelism and Yinyang filtering, so its runtime grows slowly as k increases.

Table 13. Runtime comparison and derived runtime-change index for K-Means.

Dataset	Dataset Size n	Dimension d	Cluster Number k	Spark MLlib Runtime (s)	Spark-GPU Runtime (s)
MSD	0.52 M	90	4	13.48	7.90
MSD	0.52 M	90	64	15.61	7.93
MSD	0.52 M	90	1024	39.94	8.39
MSD	0.52 M	90	10,000	547.08	11.82
Census	2.5 M	68	4	26.51	32.47
Census	2.5 M	68	64	33.76	32.77
Census	2.5 M	68	1024	63.62	33.31
Census	2.5 M	68	10,000	483.03	37.32
SuSy	5 M	18	4	255.51	188.40
SuSy	5 M	18	64	481.50	189.28
SuSy	5 M	18	1024	3830.09	191.70
SuSy	5 M	18	10,000	42,765.25	193.97
Higgs	11 M	28	4	784.73	592.42
Higgs	11 M	28	64	1642.25	602.62
Higgs	11 M	28	1024	14,066.53	607.57
Higgs	11 M	28	10,000	152,724.16	617.84

The high- k settings for SuSy and Higgs are used as scalability stress benchmarks for distance-computation throughput rather than as domain-specific classification analyses. They are included to test how SGK-Means behaves when the number of centroid comparisons becomes large and the assignment phase dominates runtime.

The results also show a low-benefit case. On the Census dataset with $k = 4$, the speedup is 0.82, meaning that SGK-Means is slower than Spark MLlib. This occurs because the arithmetic workload is too small to amortize JNI transfer, data conversion, and GPU launch overhead. When k increases to 64, the speedup becomes 1.03, and at $k = 10,000$ it reaches 12.94. This transition confirms that SGK-Means is most suitable for computation-intensive clustering rather than small-cluster configurations.

Across all datasets, Spark MLlib runtime increases significantly with k . SGK-Means runtime changes much less because GPU threads handle many sample-centroid operations in parallel and filtering removes many redundant distance evaluations. The experimental trend is consistent with the complexity analysis: as k increases, the baseline cost grows linearly in the direct assignment stage, whereas SGK-Means reduces the effective number of exact distance computations.

5.3. Influence of Dataset Size and GPU Device on SGK-Means

The dataset-size experiment uses subsets sampled from the Higgs dataset, with sizes from 0.25 M to 8 M and k fixed at 1024. Table 14 shows that both Spark MLlib and SGK-Means require more time as the dataset grows. However, the SGK-Means curve grows more slowly because the GPU assignment kernel processes samples in parallel and the communication path returns only cluster labels.

Table 14. Effect of dataset size on runtime, speedup, and runtime change.

Data Size	MLlib Time Cost (s)	Spark-GPU Time Cost (s)
0.25 M	24	3.6
0.5 M	44	5.9
1 M	73	10.6
2 M	123	22.7
4 M	223	45.8
8 M	420	119.7

For small subsets, acceleration is limited because fixed transfer and kernel-launch overheads represent a larger fraction of total runtime. For larger subsets, computation dominates and the GPU becomes more effective. The observed speedup range in this experiment is approximately 3.51 to 7.41, and the corresponding R_T values are reported in the table, showing that SGK-Means remains beneficial as sample size increases under fixed k .

Table 15 compares two GPU devices: Quadro RTX6000 (NVIDIA Corporation, Santa Clara, CA, USA) and Tesla A40 (NVIDIA Corporation, Santa Clara, CA, USA). The table reports speedups rather than raw runtimes. The purpose is to verify whether the algorithm depends on one specific GPU model. The trends are consistent across both devices: speedup increases with k , and the same datasets show similar acceleration behavior.

Table 15. Device-level acceleration comparison with derived runtime change.

Dataset	Dataset Size n	Dimension d	Cluster Number k	GPU Device
MSD	0.52 M	90	4	RTX6000/Tesla A40
MSD	0.52 M	90	64	RTX6000/Tesla A40
MSD	0.52 M	90	1024	RTX6000/Tesla A40
MSD	0.52 M	90	10,000	RTX6000/Tesla A40
Census	2.5 M	68	4	RTX6000/Tesla A40
Census	2.5 M	68	64	RTX6000/Tesla A40
Census	2.5 M	68	1024	RTX6000/Tesla A40
Census	2.5 M	68	10,000	RTX6000/Tesla A40
SuSy	5 M	18	4	RTX6000/Tesla A40
SuSy	5 M	18	64	RTX6000/Tesla A40
SuSy	5 M	18	1024	RTX6000/Tesla A40
SuSy	5 M	18	10,000	RTX6000/Tesla A40
Higgs	11 M	28	4	RTX6000/Tesla A40
Higgs	11 M	28	64	RTX6000/Tesla A40
Higgs	11 M	28	1024	RTX6000/Tesla A40
Higgs	11 M	28	10,000	RTX6000/Tesla A40

Table 15 demonstrates hardware generality and reconciles the maximum-speedup statement. In the main RTX6000 comparison of Table 13, the maximum SGK-Means speedup is 247.19. In the two-device comparison of Table 15, the Tesla A40 obtains a higher speedup of 270.15 on Higgs with $k = 10,000$. Therefore, 247.19 is reported as the maximum in the RTX6000 runtime table, whereas 270.15 is reported as the maximum observed in the hardware comparison. The stable increasing trend with k across both GPUs supports the portability of the data layout and kernel design.

The combined dataset-size and GPU-device results show that SGK-Means is controlled mainly by arithmetic intensity and communication amortization. Large k and large n increase the share of GPU-suitable computation. Very small k or small datasets may not provide enough computation to offset data transfer. This behavior is expected in

heterogeneous acceleration and should be considered when selecting SGK-Means for production workloads.

5.4. SG-Join Performance Under Partition, CPU-Core, and Dataset Factors

The SG-Join sensitivity experiments analyze partition number, CPU core count, and dataset characteristics. Table 16 reports the SG-Join and GeoSpark runtimes under different partition numbers on the Group C datasets. GeoSpark tends to become slower as partition number increases because more partitions introduce additional data movement and management overhead, while CPU computation is already close to saturation. SG-Join shows a non-monotonic pattern because partition number affects both GPU parallelism and CPU-GPU transfer overhead.

Table 16. SG-Join speedup and runtime change under different partition numbers.

Cores	Number of Partitions	SG-Join Time Cost (s)	GeoSpark Time Cost (s)
4×1	500	110	225
4×1	1000	105	290
4×1	1500	110	350
4×1	2000	130	410
4×2	500	70	87
4×2	1000	63	102
4×2	1500	70	122
4×2	2000	88	182
4×4	500	52	70
4×4	1000	43	110
4×4	1500	52	122
4×4	2000	58	165
4×8	500	43	61
4×8	1000	37	76
4×8	1500	47	89
4×8	2000	58	133

The trend in Table 16 indicates that SG-Join requires a balanced partition setting. When partitions are too few, GPU tasks are large and parallelism is insufficient. When partitions are too many, Spark scheduling and host-device transfer overhead increase. In the reported Group C experiment, a partition number around 1000 gives the best runtime in several settings. Overall, partition tuning produces speedups from approximately 1.3 to 3.2 in the partition-number experiment.

Table 17 quantifies the joint influence of partition number and CPU cores. The highest speedup in this table is 3.2 under 4×1 CPU cores and 1500 partitions. Under 4×8 CPU cores, the speedup remains positive but is lower in some settings because the CPU baseline becomes faster, while SG-Join still pays CPU-GPU transfer overhead.

Table 17. Joint effect of partition number and CPU cores on SG-Join acceleration.

CPU Cores	500 Partitions S/R_T (%)	1000 Partitions S/R_T (%)	1500 Partitions S/R_T (%)	2000 Partitions S/R_T (%)
4×1 cores	2.00/50.0	2.80/64.3	3.20/68.8	3.10/67.7
4×2 cores	1.30/23.1	1.60/37.5	1.70/41.2	2.00/50.0
4×4 cores	1.30/23.1	2.50/60.0	2.30/56.5	3.00/66.7
4×8 cores	1.40/28.6	2.10/52.4	1.90/47.4	2.30/56.5

Table 17 shows that increasing CPU cores does not eliminate the benefit of SG-Join. The speedup varies because the baseline and the accelerated method respond differently to CPU parallelism. GeoSpark executes most computation on CPU cores and therefore benefits directly from additional cores until CPU parallelism saturates. SG-Join uses CPU cores mainly for Spark-side partitioning and scheduling, while the geometric verification is executed on the GPU. Therefore, its dependency on CPU core count is weaker.

Table 18 compares the fastest runtime under each CPU-core configuration to remove the interference of non-optimal partition choices. The Spark-GPU runtime decreases relatively steadily as CPU cores increase, while the GeoSpark runtime decreases more sharply at first and then slows down. The resulting speedup first decreases and then increases slightly, but remains positive.

Table 18. Effect of CPU cores on SG-Join acceleration and runtime change.

Number of CPU Cores	SG-Join Time Cost (s)	GeoSpark Time Cost (s)
4 × 1 cores	108	274
4 × 2 cores	65	83
4 × 4 cores	42	75
4 × 8 cores	32	68

The behavior in Table 18 confirms that SG-Join is robust to CPU-core variation. Even with more CPU resources, the GPU-side filter-refinement pipeline remains faster than the CPU baseline for the tested data. The runtime advantage is not solely caused by weak CPU configuration; it comes from the combination of Spark-side partitioning, GPU-side equal-grid filtering, MBR pruning, and CUDA verification kernels.

Table 19 reports SG-Join performance across three dataset groups. Group B obtains the largest speedups, reaching 3.90 under 4 CPU cores. Group A has stable but lower speedups around 1.31 to 1.40 because the dataset is small and communication overhead occupies a larger runtime share. Group C speedups vary from 1.39 to 2.18 because the dataset is large and the spatial distribution and polygon complexity affect both filtering efficiency and verification workload.

Table 19. Dataset-level SG-Join performance and runtime-change summary.

Dataset Group	Cores Per Executor	Total CPU Cores	GeoSpark Runtime (s)	SG-Join Runtime (s)
A	8	32	15.68	11.60
B	8	32	77.39	25.66
C	8	32	62.97	35.73
A	4	16	15.69	11.20
B	4	16	103.07	33.20
C	4	16	71.71	44.82
A	2	8	15.75	11.42
B	2	8	167.26	44.89
C	2	8	87.72	63.17
A	1	4	16.60	12.67
B	1	4	268.34	68.77
C	1	4	227.75	104.56

The dataset-specific results in Table 19 show that SG-Join acceleration depends on scale, spatial distribution, overlap ratio, and polygon complexity. Group B is well matched to the GPU filter-refinement pipeline: the CPU baseline is expensive, while GPU-side filtering and verification reduce runtime significantly. Group A is too small to fully amortize GPU

transfer overhead. Group C is large, but its acceleration varies because high spatial density and irregular polygon complexity increase verification cost and load imbalance.

The SG-Join experiments demonstrate that a GPU-accelerated spatial-join operator should not use a fixed partition configuration. The optimal partition range depends on both hardware and data. Across the twelve Table 19 settings, the maximum speedup is 3.90, the geometric-mean speedup is 1.99, the arithmetic mean is 2.18, and the median is 1.68; the derived R_T values are reported alongside the individual speedups. These values are presented together because a single arithmetic average can obscure the difference between typical and stress-case acceleration.

5.5. Engineering Discussion, Validity, and Limitations

The two algorithms show different acceleration mechanisms. SGK-Means is dominated by numerical distance computation and iterative convergence. Its performance is mainly controlled by n , d , k , iteration count, and communication overhead. The single-GPU strategy is valid because it removes repeated distributed synchronization and lets the GPU execute the dominant assignment loop. The method is highly effective when k is large, but it may be ineffective when k is small and transfer overhead is comparable to computation.

SG-Join is dominated by candidate generation, spatial partition balance, MBR filtering efficiency, polygon overlap, vertex complexity, and exact verification cost. Its distributed Spark-GPU strategy is valid because spatial partitions can be processed independently. The method benefits from KD-Tree partitioning on Spark, equal-grid indexing on the GPU, and dynamic parallelism for irregular verification. It is less effective on small spatial datasets because transfer and launch overheads become significant, and it requires partition tuning to avoid underutilization or excessive data movement.

The interpretation of the experiments is bounded by the baseline structure. Spark MLib and GeoSpark/Sedona are CPU-based Spark/Sedona baselines, so the reported speedups measure end-to-end improvement over those workflows rather than superiority over all GPU systems. The results combine the effects of GPU execution, compact JNI transfer, algorithm-specific pruning, spatial partitioning, and CUDA verification kernels. For this reason, the revised manuscript reports the comparison as CPU-baseline acceleration under matched workflows and discusses GPU prior systems as related but non-identical comparators.

Tables 20–23 should be read as bounded validity and reproducibility controls rather than as unreported experiments. Table 20 provides a component-level ablation-oriented interpretation, but it is not presented as a component-removal runtime experiment. Table 22 specifies output-equivalence conditions and decision rules, but it does not introduce unarchived SSE values or join-pair counts. Table 23 therefore reports minimum, median, arithmetic mean, geometric mean, and maximum speedups so that the high- k outliers do not dominate the interpretation. Tables 24–26 further respond to the second-round review by consolidating output-equivalence requirements, retained-record boundaries, exact sign-test checks based on paired configurations, and partial component-sensitivity evidence derived from the existing tables.

Table 20. Component-level contribution and ablation-oriented interpretation.

Component	Integrated Prior Idea or Reused Mechanism	Role in the Proposed Framework	Evidence or Interpretation from Existing Results
Flattened arrays and unified indexing	Compact primitive-array communication.	Reduce JNI serialization and return labels/pair identifiers instead of full objects.	Supports wall-clock measurement that includes transfer and reconstruction overhead.
K-Means++ and Yinyang filtering	Careful seeding and bound-based pruning.	Reduce poor initialization and redundant centroid-distance evaluations.	Runtime grows slowly as k increases, especially in high-k settings.
Single Spark partition plus single GPU for SGK-Means	Algorithm-specific execution mapping.	Avoid repeated Spark shuffle and centroid synchronization in the iteration loop.	High-k cases benefit strongly, while small-k cases show limited benefit due to transfer overhead.
Spark-side KD-Tree partitioning	Spatial partitioning in distributed spatial systems.	Balance partition workloads before GPU-side refinement.	Partition-number sensitivity shows that tuning affects acceleration.
GPU equal-grid indexing and MBR filtering	Filter-refinement spatial-join design.	Prune candidate pairs before exact PIP/EI verification.	Dataset-specific results show different speedups according to scale and geometry.
CUDA dynamic parallelism	GPU-side child-grid scheduling for irregular tasks.	Mitigate irregular verification cost for complex polygon pairs.	SG-Join remains faster than GeoSpark/Sedona across the reported A-C dataset groups, but the gain varies with dataset characteristics.

Table 21. Baseline positioning against CPU Spark baselines and related GPU systems.

System or Method	Main Scope	Comparison Role in the Revised Manuscript
Spark MLlib K-Means	CPU-based distributed K-Means in Spark.	Direct SGK-Means baseline because it executes the same Spark clustering workflow.
GeoSpark/Apache Sedona	CPU-based distributed spatial processing and polygon spatial join.	Direct SG-Join baseline because it executes the same polygon spatial-join workflow.
RAPIDS/cuML	GPU machine learning and dataframe acceleration.	Closest GPU-related K-Means comparator; discussed as non-identical because the proposed method uses RDD/JNI/CUDA integration and Yinyang filtering inside a Spark-compatible operator.
GCMF and SPADE	GPU spatial-join systems and GPU-powered spatial database execution.	Closest GPU-related spatial-join comparators; discussed as related prior work because the proposed method embeds CUDA refinement into a Sedona/Spark distributed dataflow.
HeteroSpark, Spark-GPU, and GFlink	General heterogeneous big-data execution frameworks.	Used for architectural positioning rather than one-to-one runtime comparison.

Table 22. Output-consistency logic and bounded interpretation of acceleration claims.

Algorithm	Output-Consistency Point	Baseline Relation	How the Manuscript Now Uses the Claim
SGK-Means	The Yinyang filtering path preserves Lloyd-type assignment and centroid-update decisions when the same initial centroids, distance function, stopping rule, and floating-point precision are used.	Spark MLlib K-Means provides the CPU Spark baseline for runtime comparison.	The manuscript describes runtime acceleration and algorithmic equivalence conditions separately, avoiding a broad accuracy claim.
SG-Join	The final result set R contains only polygon pairs passing MBR candidate generation, exact PIP/EI verification, and duplicate removal.	GeoSpark/Apache Sedona provides the CPU spatial-join baseline using the same input layers and spatial predicate.	The manuscript clarifies that MBR filtering is not the final decision rule and that exact verification determines the output set.
Both algorithms	Runtime speedup is interpreted only after the corresponding decision rules and output definitions are specified.	The direct baselines are Spark/Sedona workflows, while GPU prior systems are positioned qualitatively.	The revised conclusion limits claims to the evaluated datasets, hardware, and protocols.

Table 23. Aggregate speedup summaries derived from the revised runtime tables.

Result Group	Source Table	Ratio Count n_s	Minimum S	Median S	Arithmetic Mean S	Geometric Mean S	Maximum S	Interpretation
SGK-Means, RTX6000	Table 13	16	0.82	2.64	36.88	6.21	247.19	Main dataset-k comparison; geometric mean is used as the typical speedup summary.
SGK-Means, two GPUs	Table 15	32	0.82	2.87	37.17	6.35	270.15	Hardware comparison; 270.15 is the maximum observed A40 speedup.
SG-Join	Table 19	12	1.31	1.68	2.18	1.99	3.90	Dataset and CPU-core comparison; speedup variation is moderate but still reported with geometric mean.

Table 24. Output equivalence, repeated-run evidence, and scope control.

Validation Item	Revised Manuscript Treatment	Evidence Boundary and Conservative Wording
SGK-Means final SSE	The manuscript now gives the strict audit rule: the two methods must use the same input data, k, distance function, initial centroids, stopping rule, and numerical precision. Final SSE should be computed as the sum of squared distances between each sample and its assigned centroid, after label-permutation effects are excluded and numerical tolerance is specified.	Final SSE values were not retained for every tested configuration. Therefore, the manuscript does not report fabricated SSE equality. The accuracy claim is replaced by an output-equivalence requirement and a correctness-condition statement.
SG-Join final join-pair set	The manuscript now specifies that final verification requires sorting deduplicated pair identifiers and comparing the SG-Join output with the GeoSpark/Sedona output under the same spatial predicate. The required empirical checks are equal final pair counts and an empty symmetric difference between the two final pair sets.	Archived join-pair count tables were not retained for every dataset group. Therefore, the manuscript does not claim numerical pair-count identity. It strengthens the exact MBR-PIP/EI-duplicate-removal decision path and states the required audit procedure.
Repeated runtime statistics	The reported times are end-to-end wall-clock aggregate measurements including data preparation, JNI conversion, host-device transfer, CUDA kernels, Spark aggregation, duplicate removal when applicable, and result reconstruction.	Five independent repeated measurements are not available for every configuration. Therefore, no unsupported standard deviations, confidence intervals, Friedman tests, Wilcoxon tests, or repeated-run <i>p</i> -values are introduced. The exact sign test in Table 25 is limited to the direction of improvement across paired configurations.
Limited paired comparison and sign-test check	The retained records allow paired comparisons across 16 SGK-Means dataset-k pairs, 12 SG-Join dataset/core pairs, and 16 SG-Join partition/core pairs. These are summarized in Table 25 using win/loss counts, minimum, median, arithmetic-mean, geometric-mean, maximum speedups, and exact two-sided sign-test <i>p</i> -values.	This strengthens the empirical response using existing quantitative data, but it remains a configuration-level directional check rather than repeated-run statistical inference. It does not establish output equivalence, runtime variance, or component-removal effects.
Partial component sensitivity	The retained records include controlled variations in k, GPU device, Spark partition number, CPU-core setting, and dataset group. Table 26 converts these into quantitative component-sensitive evidence.	This is not called a formal component-removal ablation because no Yinyang-off, dynamic-parallelism-off, MBR-off, KD-Tree-off, or GPU-baseline run was archived.
GPU-accelerated alternatives	RAPIDS/cuML, HeteroSpark, GCMF, and SPADE are discussed as related GPU systems, but not used as direct runtime baselines because their operator semantics, data abstractions, and integration scope differ from the Spark/JNI/CUDA and Sedona-compatible operators in this paper.	The manuscript makes no superiority claim over GPU alternatives and treats matched GPU-baseline experiments as future work.

Table 25. Limited paired runtime comparison and exact sign-test check based on existing dataset-parameter configurations.

Task	Existing Paired Configurations	Quantitative Paired Result and Sign-Test Check	Scope of Interpretation
SGK-Means vs. Spark MLlib	Sixteen dataset-k configurations from Table 13.	SGK-Means is faster in 15 of 16 pairs. The only non-accelerated case is Census with $k = 4$. Across the 16 pairs: minimum $S = 0.82$, median $S = 2.64$, arithmetic mean $S = 36.88$, geometric mean $S = 6.21$, and maximum $S = 247.19$. The exact two-sided sign test for 15 wins and 1 loss gives $p = 0.00052$.	This is a paired configuration-level runtime summary. The sign-test result supports a consistent direction of improvement across retained dataset-k pairs, but it is not a five-run mean and does not provide run-to-run uncertainty.
SG-Join vs. GeoSpark/Sedona	Twelve dataset-group and CPU-core configurations from Table 19.	SG-Join is faster in all 12 pairs. Across the 12 pairs: minimum $S = 1.31$, median $S = 1.68$, arithmetic mean $S = 2.18$, geometric mean $S = 1.99$, and maximum $S = 3.90$. The exact two-sided sign test for 12 wins and 0 losses gives $p = 0.00049$.	This summarizes paired workflow runtimes under the reported configurations. The p -value is a directional configuration-level check and does not replace repeated-run standard deviations or confidence intervals.
SG-Join partition/core sensitivity	Sixteen partition/core configurations from Table 17.	All reported SG-Join partition/core pairs are faster than the GeoSpark baseline. Speedups range from $S = 1.30$ to $S = 3.20$; the arithmetic mean is 2.16 and the geometric mean is 2.07. The exact two-sided sign test for 16 wins and 0 losses gives $p = 0.00003$.	This supports sensitivity to partition and CPU-core configuration while confirming that all retained paired configurations have positive speedup. It does not isolate one CUDA module and is not a component-removal ablation.

The limitations are engineering-oriented and define the scope of the reported acceleration. First, the JNI array interface reduces communication overhead but still introduces host-device transfer cost. Second, SGK-Means currently targets one GPU for the iterative loop; therefore, the largest processable dataset is bounded by single-GPU memory and by the memory required for samples, centroids, labels, and Yinyang bounds. Third, SG-Join performance depends on spatial partition quality and geometric complexity; extremely skewed datasets may require adaptive partitioning beyond static KD-Tree settings. Fourth, the framework is specialized for K-Means and polygon spatial join, and other spatial mining algorithms may require different kernels and data layouts. In addition, final SSE values, final deduplicated join-pair counts, and five-run runtime distributions were not retained for every configuration; therefore, the revision does not fabricate exact equivalence counts, standard deviations, confidence intervals, or repeated-run p -values. The exact sign-test values reported in Section 5.6 are limited to the direction of improvement across retained paired configurations and should not be interpreted as run-to-run statistical uncertainty. A matched comparison with GPU alternatives such as RAPIDS/cuML and GPU spatial-join systems is also treated as future work rather than a demonstrated superiority claim.

The central design conclusion is that algorithm characteristics determine heterogeneous mapping. Data-parallel arithmetic with frequent global synchronization should be compacted into GPU-local execution when memory capacity permits. Partition-

independent geometric workloads should be distributed by Spark and refined by GPUs. Small workloads may not amortize communication cost, while large computation-intensive workloads can expose GPU acceleration. This conclusion is a bounded engineering finding under the reported CPU-baseline protocols rather than a universal ranking over all heterogeneous systems.

Table 26. Partial component-sensitivity evidence from existing experiments.

Component Concern	Existing Controlled Variation	Quantitative Pattern from Existing Tables	Interpretation and Remaining Limitation
SGK-Means computation intensity and Yinyang/GPU distance evaluation	Cluster number k varies from 4 to 10,000 in Table 13.	The geometric-mean speedup by k increases from 1.26 at $k = 4$ to 1.94 at $k = 64$, 8.05 at $k = 1024$, and 75.58 at $k = 10,000$.	This supports the claim that acceleration is strongest when centroid-distance evaluation dominates. It does not separately isolate Yinyang filtering from raw GPU parallelism.
Single Spark partition plus single-GPU locality	Low- k and high- k cases in Table 13 expose the overhead-benefit transition.	Census with $k = 4$ has $S = 0.82$, whereas the same dataset reaches $S = 12.94$ at $k = 10,000$. SuSy and Higgs reach $S = 220.47$ and $S = 247.19$ at $k = 10,000$.	This confirms that GPU-local execution is beneficial only when arithmetic intensity amortizes transfer and launch overhead. It does not establish multi-GPU scalability.
SG-Join partitioning effect	Partition number varies from 500 to 2000 under four CPU-core settings in Table 17.	The partition/core experiment gives $S = 1.30$ to 3.20. Median speedup by partition number is 1.35 at 500, 2.30 at 1000, 2.10 at 1500, and 2.65 at 2000 partitions.	This provides partial evidence that partition choice affects the observed gain. It is not a KD-Tree-removal ablation.
Dataset-driven spatial filtering and exact refinement	Dataset groups A, B, and C vary in Table 19.	Group A shows $S = 1.31$ to 1.40, Group B shows $S = 3.02$ to 3.90, and Group C shows $S = 1.39$ to 2.18.	The different speedup ranges show that benefit depends on polygon scale, density, and refinement cost. The table does not separately disable MBR, PIP/EI, or dynamic parallelism.
CPU-core sensitivity of SG-Join	The fastest runtime under each CPU-core setting is summarized in Table 18.	The derived speedups are 2.54 for 4×1 cores, 1.28 for 4×2 cores, 1.79 for 4×4 cores, and 2.13 for 4×8 cores.	The GPU workflow remains faster under all reported core settings, but the relative advantage changes as the CPU baseline becomes stronger.

5.6. Second-Round Validation Scope and Evidence Consolidation

To address the second-round concern that explanatory tables alone are insufficient, the revised validation section now uses all quantitative records that were retained from the original experiments. The added material is deliberately limited to evidence that can be supported by the archived tables: configuration-level paired runtime comparisons, descriptive speedup summaries, exact sign-test checks for the direction of paired runtime improvement, and component-sensitive performance patterns under varied k values, GPU devices, Spark partitions, CPU-core settings, and dataset groups, as shown in Table 24. This provides a stronger quantitative response than a purely explanatory discussion, while avoiding unsupported SSE values, join-pair counts, five-run standard deviations, confidence intervals, or run-to-run statistical claims.

The output-equivalence statement is therefore treated as a precondition for correctness rather than as an inference from runtime. The manuscript now states the exact verification procedure that would be required for final SSE and final join-pair-set equivalence, and it explicitly removes any wording that could imply empirical equality without the corresponding archived numerical records.

Because each row represents a different dataset, cluster number, partition number, or CPU-core setting, Table 25 is reported as a limited paired configuration-level comparison. It summarizes how often the accelerated workflow is faster than the CPU Spark/Sedona baseline and reports minimum, median, arithmetic-mean, geometric-mean, and maximum speedups where the archived records permit these descriptive statistics. The exact two-sided sign-test p -values are added only to test the direction of paired runtime improvement across retained configurations; they do not replace repeated-run standard deviations, confidence intervals, or distributional runtime analysis.

Table 26 is presented as partial component-sensitivity evidence rather than a full component-removal ablation. The available records do not contain Yinyang-off, dynamic-parallelism-off, alternative-partitioner, or GPU-baseline experiments. Nevertheless, the reported parameter variations provide quantitative evidence that the observed speedups depend on computation intensity, partitioning, CPU-core availability, and dataset characteristics, which directly addresses the reviewer's concern that the previous ablation-oriented table was only explanatory.

6. Conclusions

6.1. Main Findings

A Spark-GPU heterogeneous acceleration framework was developed for two core spatial data mining algorithms: K-Means clustering and polygon spatial join. The design preserves Spark-side data loading, preprocessing, partitioning, scheduling, and result handling, while CUDA kernels execute the computationally dominant operations. SGK-Means uses flattened arrays, unified indexing, single Spark partition plus single-GPU execution, K-Means++ initialization, Yinyang filtering, and CUDA kernels for initialization, reassignment, centroid update, and convergence checking. SG-Join uses Apache Sedona reconstruction, Spark-side KD-Tree partitioning, JNI array communication, GPU-side equal-grid indexing, MBR filtering, PIPTest, EITest, duplicate removal, and CUDA dynamic parallelism.

The experiments show that the two algorithms benefit from different heterogeneous mappings under the evaluated CPU-baseline protocols. For SGK-Means, the RTX6000 runtime table gives a maximum speedup of 247.19 and a geometric-mean speedup of 6.21, while the two-GPU comparison gives a maximum of 270.15 on the Tesla A40. For SG-Join, the maximum speedup is 3.90 and the geometric-mean speedup is 1.99. The added exact sign-test check further supports the direction of the retained paired runtime results: SGK-Means is faster in 15 of 16 dataset- k configurations (two-sided $p = 0.00052$), SG-Join is faster in all 12 dataset/core configurations (two-sided $p = 0.00049$), and all 16 SG-Join partition/core configurations show positive speedup (two-sided $p = 0.00003$). These findings indicate that Spark-GPU acceleration is useful when the data layout, communication path, and CUDA kernel design are matched to the algorithmic bottleneck; they do not provide repeated-run uncertainty estimates and do not imply unconditional superiority over all GPU systems or all dataset regimes.

6.2. Future Work

Future work will extend the framework to additional spatial data mining algorithms, including spatial clustering variants, spatial association mining, and spatial anomaly de-

tection. The Spark-GPU communication layer will be further optimized by evaluating MPI-based or zero-copy communication mechanisms. SGK-Means will be extended to coordinated multi-GPU execution with explicit control of centroid synchronization and memory partitioning. SG-Join will be improved with adaptive spatial partitioning and scheduling strategies that respond to runtime skew, overlap density, and polygon complexity. Future evaluations should also report repeated-run dispersion, energy or cost metrics, and additional GPU baselines where execution models can be matched fairly.

In response to the remaining experimental-validity concerns, future experiments should archive final SSE values, final join-pair counts, and final pair identifiers for every configuration; rerun representative configurations at least five independent times to report means, standard deviations, and confidence intervals; add component-removal ablations such as Yinyang-off, dynamic-parallelism-off, MBR-off, and alternative partitioning; and implement matched GPU alternatives such as RAPIDS/cuML and comparable GPU spatial-join systems under identical datasets and hardware settings. The configuration-level exact sign test added in this revision should therefore be regarded as a conservative directional check based on retained paired records, not as a replacement for these future repeated-run and component-removal experiments.

Funding: This article is a preliminary result of the General Project of the Beijing Municipal Social Science Foundation (Beijing Research Center for Xi Jinping Thought on Socialism with Chinese Characteristics for a New Era), titled “Research on the Paths of Chinese Opera Art in Promoting High-Quality Development of Capital Culture” (Project No.: 23LLWXC126).

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Acknowledgments: AI-assisted language editing was used during revision to improve readability and consistency. The author reviewed, edited, and takes full responsibility for the final manuscript content.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. NVIDIA Corporation. *CUDA C++ Programming Guide, Release 13.2*; NVIDIA: Santa Clara, CA, USA, 2026.
2. NVIDIA Corporation. *RAPIDS Accelerator for Apache Spark User Guide, Release 26.04*; NVIDIA: Santa Clara, CA, USA, 2026.
3. Oracle Corporation. *Java Native Interface Specification: JDK 26*; Oracle: Austin, TX, USA, 2026.
4. Amazon Web Services. *Use the NVIDIA RAPIDS Accelerator for Apache Spark*; Amazon EMR Release Guide; Amazon Web Services: Seattle, WA, USA, 2025.
5. Eldawy, A.; Mokbel, M.F. SpatialHadoop: A MapReduce Framework for Spatial Data. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Republic of Korea, 13–17 April 2015; pp. 1352–1363. [\[CrossRef\]](#)
6. Apache Software Foundation. *Spatial RDD Application: Apache Sedona Documentation*; Apache Sedona: Wilmington, DE, USA, 2026.
7. Apache Software Foundation. *Spatial DataFrame/SQL Application: Apache Sedona Documentation*; Apache Sedona: Wilmington, DE, USA, 2026.
8. Google Cloud. *Earth Engine*; Google Cloud: Mountain View, CA, USA, 2026.
9. Microsoft. *Azure Maps Documentation*; Microsoft Learn: Redmond, WA, USA, 2025.
10. Sentinel Hub. *Sentinel Hub API Documentation*; Sinergise: Ljubljana, Slovenia, 2026.
11. Li, P.; Luo, Y.; Zhang, N.; Cao, Y. HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms. In Proceedings of the 2015 IEEE International Conference on Networking, Architecture and Storage, Boston, MA, USA, 6–7 August 2015; pp. 347–348. [\[CrossRef\]](#)
12. Yuan, Y.; Salmi, M.F.; Huai, Y.; Wang, K.; Lee, R.; Zhang, X. Spark-GPU: An Accelerated In-Memory Data Processing Engine on Clusters. In Proceedings of the 2016 IEEE International Conference on Big Data, Washington, DC, USA, 5–8 December 2016; pp. 273–283. [\[CrossRef\]](#)
13. Chen, C.; Li, K.; Ouyang, A.; Zeng, Z.; Li, K. GFlink: An In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 1275–1288.
14. RAPIDS. *RAPIDS: GPU-Accelerated Data Science*; RAPIDS AI: Santa Clara, CA, USA, 2026.

15. Lloyd, S. Least Squares Quantization in PCM. *IEEE Trans. Inf. Theory* **1982**, *28*, 129–137. [\[CrossRef\]](#)
16. Arthur, D.; Vassilvitskii, S. K-Means++: The Advantages of Careful Seeding. In Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, USA, 7–9 January 2007; pp. 1027–1035.
17. Sardar, T.H.; Ansari, Z. An Analysis of MapReduce Efficiency in Document Clustering Using Parallel K-Means Algorithm. *Future Comput. Inform. J.* **2018**, *3*, 200–209. [\[CrossRef\]](#)
18. Taha, R.; Alshakrani, S.; Alqaddoumi, A. Implementing Parallel Computing to Enhance the Performance of K-Mean Algorithm. In Proceedings of the 2021 International Conference on Data Analytics for Business and Industry, Sakheer, Bahrain, 25–26 October 2021; pp. 140–143. [\[CrossRef\]](#)
19. Dias, L.A.; Ferreira, J.C.; Fernandes, M.A.C. Parallel Implementation of K-Means Algorithm on FPGA. *IEEE Access* **2020**, *8*, 41071–41084. [\[CrossRef\]](#)
20. Cuomo, S.; De Angelis, V.; Farina, G.; Marcellino, L.; Toraldo, G. A GPU-Accelerated Parallel K-Means Algorithm. *Comput. Electr. Eng.* **2019**, *75*, 262–274. [\[CrossRef\]](#)
21. Lapegna, M.; Balzano, W.; Meyer, N.; Romano, D. Clustering Algorithms on Low-Power and High-Performance Devices for Edge Computing Environments. *Sensors* **2021**, *21*, 5395. [\[CrossRef\]](#) [\[PubMed\]](#)
22. Apache Software Foundation. *KMeans: PySpark Machine Learning Clustering API*; Apache Spark: Wilmington, DE, USA, 2026.
23. Hughes, J.N.; Annex, A.; Eichelberger, C.N.; Fox, A.; Hulbert, A.; Ronquest, M. GeoMesa: A Distributed Architecture for Spatio-Temporal Fusion. In *Proceedings of SPIE Geospatial Informatics, Fusion, and Motion Video Analytics V*, Baltimore, MD, USA; SPIE: Bellingham, WA, USA, 2015; Volume 9473, p. 94730F. [\[CrossRef\]](#)
24. Aghajarian, D.; Puri, S.; Prasad, S.K. GCMF: An Efficient End-to-End Spatial Join System over Large Polygonal Datasets on GPGPU Platform. In Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, San Francisco, CA, USA, 31 October–3 November 2016; p. 18. [\[CrossRef\]](#)
25. Doraiswamy, H.; Freire, J. SPADE: GPU-Powered Spatial Database Engine for Commodity Hardware. In Proceedings of the 2022 IEEE 38th International Conference on Data Engineering, Kuala Lumpur, Malaysia, 9–12 May 2022; pp. 2669–2681. [\[CrossRef\]](#)
26. Georgiadis, T.; Mamoulis, N. Raster Intervals: An Approximation Technique for Polygon Intersection Joins. In Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data, Seattle, WA, USA, 18–23 June 2023; p. 40. [\[CrossRef\]](#)
27. Chen, J.; Zhang, Q.; Han, P.; Zhang, Y.; Lu, Y.; Zheng, P. Energy-aware Scheduling of Workflow Applications Towards Schedule Length Optimization in Heterogeneous Distributed Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.* **2025**, *31*, 2. [\[CrossRef\]](#)
28. He, S.; Wang, T.; Spicer, R.A.; Farnsworth, A.; Mulch, A.; Widdowson, M.; Zhang, Q.; Cai, F.; Valdes, P.J.; Wang, C.; et al. Back to an ice-free future: Early Cretaceous seasonal cycles of sea surface temperature and glacier ice. *Sci. Adv.* **2025**, *11*, eadr9417. [\[CrossRef\]](#) [\[PubMed\]](#)
29. Lv, J.; Wang, C.; Xie, L. Adaptive distributed observer design for nonlinear multiagent systems. *Automatica* **2026**, *183*, 112625. [\[CrossRef\]](#)
30. Lv, J.; Wang, C.; Xie, L. Distributed output-feedback leader-following consensus of nonlinear multiagent systems. *Automatica* **2025**, *177*, 112281. [\[CrossRef\]](#)
31. Wang, Y.-N.; Li, S.-K.; Li, Z.-Y.; Garg, A. Exploring the application of the MICP technique for the suppression of erosion in granite residual soil in Shantou using a rainfall erosion simulator. *Acta Geotech.* **2023**, *18*, 3273–3285. [\[CrossRef\]](#)
32. Shen, T.; Cai, Y.; Liu, L. Green supply chain with strategic retailers: Transportation-based carbon penalty. *Energy Econ.* **2025**, *145*, 108411. [\[CrossRef\]](#)
33. Huang, X.; Kang, N.; Wang, Q.; El Mansori, M.; Guittonneau, F. Wear of directed energy deposited H13 steel as a function of its graded microstructure. *Tribol. Int.* **2025**, *207*, 110617. [\[CrossRef\]](#)
34. Li, Y.; Qu, K.; Jiang, R.; Wang, H.; Zhao, X.; Yang, Z.; Tian, B.; Tao, J.; Chu, J.; Duan, C. Atomic-Scale Defect Reconfiguration via Thermally Induced Structural Ordering for High-Efficiency Sb₂Se₃ Solar Cells. *ACS Nano* **2025**, *19*, 33460–33472. [\[CrossRef\]](#) [\[PubMed\]](#)
35. Lai, C.-H.; Wu, T.-E.; Wang, C.-C. Enhancing Information Security in Smart Manufacturing Through Least Significant Bit Steganography in Engineering Drawings. *J. Comput. Inf. Sci. Eng.* **2025**, *25*, 091006. [\[CrossRef\]](#)
36. Chen, P.; Nie, X.; Ning, Y.; Zhang, Y. Learning Efficient and Adaptive Cross-Channel Dependencies for Weakly-Supervised Object Detection. *IEEE Trans. Multimed.* **2025**, *27*, 8954–8966. [\[CrossRef\]](#)
37. Wang, L.; Ma, Y.; Yan, Z.; Zhang, L.; Hu, Y.; Zhao, S. Giving or receiving: Impact of online socializing in online fitness community on physical activity and emotional state. *Comput. Hum. Behav.* **2025**, *169*, 108669. [\[CrossRef\]](#)
38. Wang, L.; Bala, H.; Yan, L.; Guo, X. Physicians' Contributions to Online Healthcare Platforms: Relative Effects of Herding Cues and Feedback Types. *J. Manag. Inf. Syst.* **2026**, *43*, 205–236. [\[CrossRef\]](#)
39. Zhang, L.; Wu, T. Solving SAT by Restricted Spiking Neural P Systems with Temporal Division Rules. *Theor. Comput. Sci.* **2026**, *1078*, 116009. [\[CrossRef\]](#)

40. Bu, Z.; Mo, D.; Sun, Y.; Zhang, L.; Zhu, Y.; Duan, X.; Nian, L.; Tang, Y.; Ye, C.; Cao, J. Dynamic Accumulation and Co-occurrence of Phthalates and Squalene in Settled Dust: Implications for Assessing Indoor Phthalate Homology and Exposures. *Environ. Sci. Process. Impacts* **2026**, *28*, 1721–1732. [CrossRef] [PubMed]
41. Li, Y.; Zhang, X.; He, Z.; Xie, J.; Zhang, Y.; Wang, J.; Deng, J. Response of sludge EPS under the action of coupled alternating electric field-ammonium persulfate dewatering technology: Structural changes and mechanism analysis. *Chem. Eng. Sci.* **2025**, *323*, 123267. [CrossRef]
42. Li, K.; Shen, P.; Cheng, L.; Chen, W.; Wang, L.; Gao, J.; Yu, M.; Zhang, Z.; Xie, Y.; Lu, L. Multiphysics-driven surgical electrode-tissue interactions: Phenomena, mechanisms, and insights. *Int. J. Mech. Sci.* **2026**, *311*, 111179. [CrossRef]
43. Tong, T.; Qu, W.; Hua, J.; Chen, D.; Tan, J.; Lin, J. Delamination detection in composite laminates using Lamb wave tomographic method based on sparse and probabilistic reconstruction. *NDT E Int.* **2026**, *160*, 103650. [CrossRef]
44. Weng, Y.; Yang, K.; Liu, Z.; He, W.; Tang, X. Lgvlm-miot: A lightweight generative visual-language model for multilingual iot applications. *IEEE Internet Things J.* **2025**, *12*, 13311–13326. [CrossRef]
45. Yang, T.; Qian, Z.; Hang, N.; Liu, M. S-PINN: Stabilized physics-informed neural networks for alleviating barriers between multi-level co-optimization. *Comput. Methods Appl. Mech. Eng.* **2025**, *447*, 118348. [CrossRef]
46. Ye, L.; Li, Z.-H.; Yan, H.-N.; Liu, C.; Cho, H.H.; Guo, T. Predicting Film-Cooling Effectiveness of Compound-Angle Holes Using a POD-Based Hybrid Deep Learning Model. *Aerosp. Sci. Technol.* **2026**, *176*, 112590. [CrossRef]
47. Li, B.-R.; Liu, C.-L.; Liu, P.-J.; Zhou, T.-L.; Xu, W.-J.; Wang, L.-Q.; Ye, L. Predictive analysis of the overall cooling effectiveness for leading-edge showerhead cooling based on a thermal resistance network model with physical corrections. *Aerosp. Sci. Technol.* **2026**, *176*, 112492. [CrossRef]
48. Hao, Z.; Zhang, Y.; Zhu, Y.; Sun, B.; Jiao, H.; Hu, Y.; Lin, Z.; Feng, L.; Wang, A.; Xiao, S.; et al. Compound metalens-based miniature two-photon microscope for large-FOV imaging in freely behaving animals. *Photonix* **2025**, *6*, 57. [CrossRef]
49. Wei, J.; Chen, Z.; Zhang, Y.; Zhang, W.; Li, J.; Lora, E.E.S.; Venturini, O.J.; Kozlov, A.N.; Isa, Y.M. An adjustable heat-to-power ratio combined heat and power system integrating biomass gasification, solid oxide fuel cell, and gas turbine: Energy, exergy, economic, and exergoeconomic (4E) analyses. *Int. J. Hydrogen Energy* **2026**, *244*, 155706. [CrossRef]
50. Fan, M.; Zhang, N.; Su, D.; Deng, W.; Zhang, R.-X.; Cai, G.-J.; Chen, X.-S. A Comprehensive Investigation of Particle Gradation Effects on Limiting Void Ratios and Pore Structures of Granular Soils. *Int. J. Numer. Anal. Methods Geomech.* **2026**, *50*, 3998–4019. [CrossRef]
51. Cao, Y.; Liu, Y.; Xie, H.; Kong, X.; Cong, T.; Gu, X.; Xi, Z. An electrothermally actuated high-power MOEMS optical switch with a state-latching mechanism. *Appl. Phys. Lett.* **2026**, *128*, 173303. [CrossRef]
52. Roy, T.; Zhou, Y. From agricultural waste to building insulation: A chemical-free composite from banana stem and jackfruit seed starch. *Bioresour. Technol.* **2026**, *452*, 134591. [CrossRef] [PubMed]
53. Ni, L.; Wu, S.; Pu, J.; Xin, S.; Wang, A.; Chen, X.; Lai, Z.; Meng, Y. Highly reliable RRAM-based physical unclonable function with auto-write technique. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2026**, *34*, 2000–2004. [CrossRef]
54. Guo, X.; Zhang, Y.; Liu, B.; Gong, W.; Ren, H.; Wang, X.; Li, Y.; Luo, J.; Wang, C. Lu Single Atoms Create Long-Lived Interlayer Charge Highways for Efficient Visible-Light CO₂ Photoconversion. *Appl. Catal. B Environ. Energy* **2026**, *398*, 127024. [CrossRef]
55. Lin, H.; Basem, A.; Yang, Y.; Alkhayyat, A.; Alhumaid, S.; Zhu, P.; Alotaibi, T.S.; Jaghdam, I.H.; Alkhalaf, S.; Ali, H.E. Leveraging cascade heat recovery from a gas turbine cycle in a sustainable integrated energy system; multi-objective optimization using radial basis function-driven Ant Colony algorithm. *Int. Commun. Heat Mass Transf.* **2026**, *174*, 110962. [CrossRef]
56. Chen, X.; Huang, J.; Zhu, J.; Jing, J.; Hu, Z.; Bu, X.; Fan, J.; Li, S.; Song, W.; Han, X.; et al. An Amorphous Polyimide-Based Positive Electrode for High-Capacity and Durable Aluminum Dual-Ion Batteries. *Angew. Chem. Int. Ed.* **2026**, *65*, e2153127. [CrossRef] [PubMed]
57. Wang, Z.; Wang, X.; Xu, S.; Zhou, R.; Zhang, M.; Li, W.; Zhang, Z.; Wang, L.; Chen, J.; Zhang, J.; et al. Off-site production of plasma-activated water for efficient disinfection: The crucial role of high valence NO_x and new chemical pathways. *Water Res.* **2024**, *267*, 122541. [CrossRef] [PubMed]
58. Fan, R.; Boukerche, A.; Pan, P.; Jin, Z.; Su, Y. An Underwater Secure Localization Scheme Based on Physical Layer Cryptographic Learning. *IEEE Trans. Mob. Comput.* **2025**, *25*, 37–53. [CrossRef]
59. Zhou, F.; Wang, Z.; Zhao, S.; Zhang, X.; Guo, Q.; Dai, R.; Meng, F.; Lyu, H.; Wan, J.; Dong, Y.; et al. Data-driven discovery of high-performance surface passivators for efficient and stable perovskite solar cells. *Chem. Eng. J.* **2026**, *535*, 175529. [CrossRef]
60. Yu, G.; Tang, X.; Deng, L.; Jin, J.; Gong, P.; Zhang, M.; Wang, X.; Li, Y. A general data-driven framework for fully convex anisotropic yield criterion modeling. *Int. J. Plast.* **2026**, *203*, 104735. [CrossRef]
61. Chen, L. Beyond External Constraints: The Missing Dimension of AI Governance (Working Paper). SSRN. 2026. Available online: <https://ssrn.com/abstract=6449738> (accessed on 20 May 2026).
62. Xi, W.; Ping, Y. Bibliometric analysis of global pyrite research: Trends, collaborations, and emerging frontiers (2005–2024). *Earth Sci. Inf.* **2026**, *19*, 52. [CrossRef]

63. Xi, W.; Ping, Y.; Cai, H.; Tan, Q. Levels, Distribution, and Sources of Heavy Metals in Tobacco-Growing Soils of Liancheng County: Insights From PCA and PMF Models. *Int. J. Anal. Chem.* **2026**, *2026*, 6078185. [[CrossRef](#)]
64. Xi, W.; Ping, Y.; Cai, H.; Tan, Q. Source Identification and Apportionment of Heavy Metals in Tobacco Farmland Soils via Positive Matrix Factorization and Geochemical Indices. *Pol. J. Environ. Stud.* **2026**. [[CrossRef](#)] [[PubMed](#)]
65. Zhai, S.B.; Cheng, J.M.; Li, G.W.; Wang, Z.Y.; Xing, Y.M.; Hou, M.S. Robust auto-landing guidance and control for fixed-wing unmanned aerial vehicles with strong wind disturbances. *Aerosp. Sci. Technol.* **2026**, *178*, 112413. [[CrossRef](#)]
66. Zhai, S.B.; Li, G.W.; Wu, G.; Nan, K.Z.; Zhang, H.Z.; Hou, M.S. Cooperative 4D trajectory planning for multi aerial vehicles combining receding horizon optimization and differential evolution algorithm in dynamic battlefield environment. *Appl. Soft Comput.* **2026**, *197*, 115169. [[CrossRef](#)]
67. Wang, Z.; Gong, T.; Nian, Y.; Yi, B.; Wang, X.; Zhou, X.; Lv, J.; Min, G.; Li, K. SFTRAP: Satisfying Fidelity Threshold Routing and Adaptive Purification for Throughput Maximum in Quantum Network. *IEEE Trans. Commun.* **2026**, *74*, 3952–3967. [[CrossRef](#)]
68. Wang, H.; Wang, Y.; Li, Y.; Wang, X.; Chen, X.; Yu, S.; Wang, X. Spatial Risk Prediction and Serological Validation of Emerging Wild Boar-Borne Diseases in Eastern Heilongjiang, China. *Pak. Vet. J.* **2025**, *45*, 1721–1732. [[CrossRef](#)]
69. Xu, J.; Tang, R.; Lv, P.; Yu, M.; Yu, G.; Chen, E. LTKT: Knowledge Tracing Based on Positive and Negative Learning Transfers. *Tsinghua Sci. Technol.* **2026**, *31*, 1894–1917. [[CrossRef](#)]
70. Zhang, M.; Yang, Z.; Ma, Y.; Huang, T.; Li, Q.; Gong, P.; Deng, L.; Jin, J.; Liu, J.; Guo, P.; et al. Oxide growth and surface fracture during high-temperature oxidation of Zr55Cu30Al10Ni5 metallic glass powders. *Appl. Surf. Sci.* **2026**, *736*, 166816. [[CrossRef](#)]
71. Li, Y.; Jin, Z.; Ahmed, S.U.; Das, J.; Bleher, R.; Chang, D.; Yu, X.; Flynn, C.D.; Zhou, J.; Hu, X.; et al. Biomolecular Condensates as Protein Degradation Tools for Intracellular Targets. *Nat. Commun.* **2026**. [[CrossRef](#)] [[PubMed](#)]
72. Fan, M.; Su, D.; Zhang, N.; Cai, G.-J.; Chen, X.-S. Coupled effects of particle multilevel morphology and intermediate principal stress ratio on the macro-and micro-mechanical behavior of granular soils: A DEM study. *Transp. Geotech.* **2026**, *62*, 102142. [[CrossRef](#)]
73. Zhang, Z.-Z.; Zhang, L.-H.; Li, H.; Chen, F.; Nikolaevna, D.S.; Li, M.-H.; Liu, B.-F.; Li, Y.-F.; Zhang, Z.-F. Is BPA only the tip of the iceberg? Occurrence and risk assessment of 32 bisphenol analogues in landfill leachate. *J. Hazard. Mater.* **2026**, *512*, 142403. [[CrossRef](#)] [[PubMed](#)]
74. Xie, J.; Jiang, Z.; Wu, Z.; Shen, J.; Zhou, Y.; Chen, Y.; He, P.; Oliveira, J.P. Joining of TiAl intermetallic Compound/CoCrFeMnNi high-entropy alloy dissimilar metals by vacuum brazing process: Microstructure, mechanical properties and processing parameters. *Intermetallics* **2026**, *197*, 109374. [[CrossRef](#)]
75. Chen, X.; Feng, C.; Rong, W.; Kwon, M. From ‘Ease of Use’ to ‘Emotional Attachment’: How Service Quality of O2O Platforms Drives Consumer Emotions and Continued Usage in the Digital Food Delivery Era. *Acta Psychol.* **2026**, *268*, 107224. [[CrossRef](#)] [[PubMed](#)]
76. Han, X.; Qin, Z.; Lin, Y.; Chang, Y.; Guzmán-Iñigo, J.; Zhang, C. Experimental Investigation on Thermoacoustic and Structural Dynamics in a Premixed Arrayed Micro-Tube Hydrogen Combustor. *Engineering* **2026**. [[CrossRef](#)]
77. Zhu, W.; Xu, B.; Guo, B.; Xie, C.; Xiong, R. Data-efficient and Interpretable Long-term Prognostics of PEMFC Degradation via Physics-constrained Symbolic Regression–Physics-informed Neural Network. *eTransportation* **2026**, *29*, 100615. [[CrossRef](#)]
78. Wen, L.; Hu, D.; Zhang, R.; Hu, X.; Luo, Y.; Chen, H.; Liang, Y.; Li, Y.; Wang, S. Two-dimensional SnSe2/Bi2Se3 vertically stacked heterostructures for enhanced photoresponse through tuning layer coverage. *iScience* **2026**, *29*, 115392. [[CrossRef](#)] [[PubMed](#)]
79. Chen, C.; Xia, H.; Zhou, W.; Xia, C.; Liu, M.; Hao, R.; Wang, T. CASET: A cascaded attention-based framework for semantic explainability of toxicity in large language models. *Appl. Intell.* **2026**, *56*, 196. [[CrossRef](#)]
80. Zhang, Y.; Yuqi, Z.; Liang, Y.; Mu, S.; Zhang, Z.; Chen, X. AD-VGF: An Improved Generative Adversarial Network for Credit Card Risk Identification and Management in Digital Economy. *Cybern. Syst.* **2025**, *56*, 1–26. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.