High-Performance Polyline Intersection based Spatial Join on GPU-Accelerated Clusters

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, 10016
syou@gradcenter.cuny.edu

Jianting Zhang
Dept. of Computer Science
City College of New York
New York City, NY, 10031
jzhang@cs.ccny.cuny.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73071
ggruenwald@ou.edu

Abstract
The rapid growing volumes of spatial data have brought significant challenges on developing high-performance spatial data processing techniques in parallel and distributed computing environments. Spatial joins are important data management techniques in gaining insights from large-scale geospatial data. While several distributed spatial join techniques based on spatial partitions have been implemented on top of existing Big Data systems, they are not capable of natively exploiting massively data parallel computing power provided by modern commodity Graphics Processing Units (GPUs). In this study, as an important component of our research initiative in developing high-performance spatial join techniques on GPUs, we have designed and implemented a polyline intersection based spatial join technique that is capable of exploiting massively data parallel computing power on GPUs. The proposed polyline intersection based spatial join technique is integrated into a customized lightweight distributed execution engine that natively supports spatial partitions. We empirically evaluate the performance of the proposed spatial join technique on both a standalone GPU-equipped workstation and Amazon EC2 GPU-accelerated clusters and demonstrate its high performance when comparing with the state-of-the-art.

Categories and Subject Descriptors
H.2.8 [**Database Management**]: Database Applications – *Spatial databases and GIS.*

General Terms
Performance, Design, Experimentation.

Keywords
Spatial Join, Big Data, Polyline Intersection, Data Parallel, GPU

1. INTRODUCTION
Advances of sensing, modeling and navigation technologies and newly emerging applications, such as satellite imagery for Earth observation, environmental modeling for climate change studies and GPS data for location dependent services, have generated large volumes of geospatial data. Very often multiple spatial datasets need to be joined to derive new information to support decision making. For example, for each pickup location of a taxi trip record, a point-in-polygon test based spatial join can find the census block that a point falls within. Time-varying statistics on the taxi trips originated and ended at the census blocks can potentially reveal travel and traffic patterns that are useful for city and traffic planning. As another example, for each polygon boundary of a US Census Bureau TIGER record, a polyline intersection based spatial join can find river network (or *linearwater*) segments that it intersects. While traditional Spatial Databases and Geographical Information Systems (GIS) have provided decent supports on spatial joins for small datasets, the performance is not acceptable when the data volumes are large. It is thus desirable to speed up spatial join query processing in distributed computer clusters. As spatial joins are typically both data intensive and computing intensive and Cloud computing facilities are increasingly equipped with modern multi-core CPUs, many-core GPUs and large memory capacity[i], new computing techniques that are capable of effectively utilizing modern parallel and distributed platforms are both technically preferable and practically useful.

Several pioneering distributed spatial data management systems, such as HadoopGIS [1] and SpatialHadoop [2], have been developed on top of the Hadoop platform and have achieved impressive scalability. More recent developments, such as SpatialSpark [3], GeoSpark [4], Simba [5] and ISP [6], are built on top of in-memory systems, such as Apache Spark [7] and Cloudera Impala [8], with both efficiency and scalability. We refer to Section 2 for more discussions on the distributed spatial join techniques and the respective research prototype systems. While spatial joins on multi-core CPUs can naturally utilize open source geometry libraries (such as JTS and GEOS) and support all major spatial operators, including point-in-polygon test, point-to-polyline distance and polyline/polygon intersection test, supporting such spatial operators on GPUs is technically challenging. Efficient spatial indexing techniques and several spatial operators for spatial join query processing on GPUs have been developed over the past few years [9] [10]. However, to the best of our knowledge, there is no existing work on supporting polyline intersection based spatial join on GPU-equipped clusters, which motivates this study.

In our previous work, although we were able to demonstrate that ISP-MC+ and ISP-GPU are capable of achieving much higher efficiency by natively exploiting the parallel processing power of multi-core CPUs and GPUs [6], respectively, they were developed on top of Cloudera Impala, a Big Data system for relational data, and supported broadcast one side to all partitions in the other side of a relational join only. The restriction has made it very cumbersome (if not impossible) to spatially partition and perform spatial joins on both sides. This is mainly because Impala is designed as an end-to-end system which is different from Hadoop and Spark that are designed as platforms for easy extensions, such as the developments of SpatialHadoop and SpatialSpark. As such, in this study, we aim at developing a partition-based spatial join technique by extending the LDE engine that we have developed previously [11] for distributed large-scale spatial data processing and evaluating its performance using real world datasets.

5th ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data (BigSpatial'16) 978-1-4503-4581-1

Our technical contributions are three-fold. First, we have developed efficient data parallel designs and implementations on GPUs for polyline intersection based spatial joins, which also work for multi-core CPUs. Second, we have seamlessly integrated the designs and implementations with a spatial-aware lightweight distributed execution engine to achieve high performance. Third, by comparing with existing systems such as SpatialHadoop using publically available large-scale geospatial datasets, we have demonstrated that our techniques that are capable of natively exploit parallel computing power of GPU-accelerated clusters can achieve significantly higher performance, due to the improvements of both parallel geometry operations for polyline intersection test and distributed computing infrastructure for in-memory processing.

The rest of the paper is organized as follows. Section 2 provides background, motivation and related work. Section 3 introduces partition-based spatial joins and its implications in distributed spatial join query processing. Section 4 is the system architecture and design and implementation details for polyline intersection based spatial joins. Section 5 reports experiments and their results. Finally, Section 6 is the conclusion and future work directions.

2. BACKGROUND, MOTIVATION AND RELATED WORK

Spatial join is a well-studied topic in Spatial Databases and we refer to [12] for an excellent survey in traditional serial, parallel and distributed computing settings. A spatial join typically has two phases: the filtering phase and the refinement phase. The filtering phase pairs up spatial data items based on their Minimum Bounding Rectangle (MBR) approximation by using either pre-built or on-the-fly constructed spatial index. The refinement phase applies computational geometry algorithms to filter out pairs that do not satisfy the required spatial criteria in the spatial join, typically in the form of a spatial predicate, such as point-in-polygon test or polyline intersection test. While most of the existing spatial join techniques and software implementations are based on serial computing on a single computing node, techniques for parallel and distributed spatial joins have been proposed in the past few decades for different architectures [12]. Although these techniques differ significantly, a parallel spatial join typically has additional steps to partition input spatial datasets (spatially and non-spatially) and join the partitions globally (i.e., global join) before data items in partition pairs are joined locally (i.e., local join).

As Hadoop-based Cloud computing platforms become mainstream, several research prototypes have been developed to support spatial data management, including spatial joins, on Hadoop. HadoopGIS [1] and SpatialHadoop [2] are among the leading works on supporting spatial data management by extending Hadoop. We have also extended Apache Spark for spatial joins and developed SpatialSpark [3]. Since Simba [5] and GeoSpark [4] seem to support spatial joins on point data only (point-to-point distance based join), we next focus on comparing HadoopGIS, SpatialHadoop and SpatialSpark that support spatial joins on major types of spatial data (point/polyline/polygon) by using geometry libraries (such as JTS and GEOS). HadoopGIS adopts the Hadoop Streaming[ii] framework and uses additional MapReduce jobs to shuffle data items that are spatially close to each other into the same partitions before a final MapReduce job is launched to process reorganized data items in the partitions. SpatialHadoop extends Hadoop at a lower level and has random accesses to both raw and derived data stored in the Hadoop Distributed File System (HDFS[iii]). By extending *FileInputFormat* defined by the Hadoop runtime library, SpatialHadoop is able to spatially index input datasets, explicitly access the resulting index structures stored in HDFS and query the indexes to pair up partitions based on the index structures, before a *Map*-only job is launched to process the pairs of partitions in distributed computing nodes. SpatialSpark is based on Apache Spark. Spark provides an excellent development platform by automatically distributing tasks to computing nodes, as long as developers can express their applications as data parallel operations on collection/vector data structures, i.e., Resilient Distributed Datasets (RDDs) [7]. The automatic distribution is based on the key-value pairs of RDDs which largely separate domain logic from parallelization/distribution. A more detailed review of the three research prototype systems and their performance comparisons using public datasets are reported in [13].

Different from SpatialHadoop, HadoopGIS and SpatialSpark are forced to access data sequentially within data partitions due to the restrictions of the underlying platform (Hadoop Streaming for HadoopGIS and Spark RDD for SpatialSpark). The restrictions, due to the streaming data model (for HadoopGIS) and Scala functional programming language (for SpatialSpark), have significantly lower the capabilities of the two systems in efficiently supporting spatial indexing and indexed query processing. Indeed, spatial indexing in the two systems is limited to intra-partitions and requires on-the-fly reconstructions from raw data. The implementations of spatial joins on two datasets are conceptually cumbersome as partition boundary is invisible and cross-partition data reference is supported by neither Hadoop Streaming nor Spark RDD. To solve the problem, both HadoopGIS and SpatialSpark require additional steps to globally pair up partitions based on spatial intersections before parallel/distributed local joins on individual partitions. While the additional steps in SpatialSpark are implemented as two *GroupBy* primitives in Spark which are efficient for in-memory processing, they have to be implemented as multiple MapReduce jobs in HadoopGIS and significant data movements across distributed computing nodes (including Map, Reduce and Shuffle phases) are unavoidable. The excessive disk I/Os are very expensive. On the other hand, while SpatialHadoop has support on storing, accessing and indexing geometric data in binary formats with random access capabilities by significantly extending the Hadoop runtime library, its performance is considerably limited by Hadoop and is inferior to SpatialSpark for data-intensive applications, largely due to the performance gap between disk and memory. Note that we have deferred the discussions on spatial partitioning in the three systems to Section 3.

As HadoopGIS and SpatialHadoop are based on Hadoop and SpatialSpark is based on Spark - the two systems that are based on either Java or Scala programming language and both rely on Java Virtual Machines (JVMs) - native parallel programming tools which are likely to help achieve higher performance cannot be easily incorporated. Furthermore, currently JVMs support Single Instruction Multiple Data (SIMD) computing power on neither multi-core CPUs nor GPUs. HadoopGIS has attempted to integrate GPUs into its MapReduce/Hadoop framework. While the underlying GPU-accelerated PixelBox geometry library [9] is efficient, the performance gain was not significant [14], likely due to the mismatch between fast GPU processing and dominating slow Hadoop IOs. A framework on using GPUs for spatial computing has also been proposed by the HadoopGIS team [15]. However, their current effort mostly focused on evaluating tree indexing structures and filtering and their prototype system does not support end-to-end spatial joins yet.

To effectively utilize the increasingly important SIMD computing power, we have extended the leading open source in-memory Big Data system Impala [8] that has a C++ backend to support spatial joins using native parallel processing tools, including OpenMP, Intel Thread Building Blocks (TBB[iv]) and Compute Unified Device Architecture (CUDA[v]). By extending block-based join in Impala, our In-memory Spatial Processing (ISP) framework [6] is able to accept a spatial join in a SQL statement, parse the data in the two sides of a spatial join in chunks (row-batches), build spatial index on-the-fly to speed up local joins in chunks. While ISP employs the SQL query interface inherited from Impala, its current architecture also limits our extension to broadcast-based spatial join, i.e., broadcasting the whole dataset on the right side of a join to chunks of the left side of the join for local join (termed as left-partition-right-broadcast). As Impala is designed for relational data, in order to support spatial data under the architecture, ISP was forced to represent geometry as strings in the Well-Know-Text (WKT[vi]) format. In a way similar to HadoopGIS, this increases not only data volumes (and hence disk I/Os) significantly, but also infrastructure overheads. The simple reason is that text needs to be parsed before geometry can be used and intermediate binary geometry needs to be converted to strings for outputting.

Our recent work on developing the Lightweight Distributed Execution (LDE) engine for spatial joins aims at overcoming these disadvantages by allowing accessing HDFS files (including both data and index) randomly in a principled way [11]. Experiments have demonstrated the efficiency of LDE-MC+ and LDE-GPU when compared with ISP-MC+ on multi-core CPUs and ISP-GPU on GPUs, respectively [11]. When comparing ISP-MC+ with ISP-MC, a baseline implementation of spatial join using the C/C++ based GEOS geometry library, ISP-MC+ is much more efficient than ISP-MC [6], which demonstrates the importance of the efficiency of the underlying geometry library in end-to-end performance of large-scale spatial join query processing. On the other hand, while ISP-MC+, ISP-GPU, LDE-MC+ and LDE-GPU support point-in-polygon test and point-to-polyline distance based spatial joins, they do not support polyline intersection based spatial join yet, which motives this study. We note that HadoopGIS also uses GEOS library within the wrapped reducer for local join as ISP-MC does. However, we found that the C++ based GEOS library is typically several times slower than its Java counterpart (JTS library) which makes ISP-MC unattractive when comparing ISP-MC with SpatialSpark [3]. We suspect that this is also one of the important factors that contributes to SpatialHadoop's superior performance when comparing the end-to-end performance of HadoopGIS and SpatialHadoop as reported in [13].

The developments of ISP and LDE are driven by the practical needs of spatially joining GPS point locations with urban infrastructure data or global ecological zone data based on several spatial join criteria, including point-in-polygon test and point-to-polyline distance. This makes broadcast-based spatial join a natural choice in ISP based on Impala as broadcast is natively supported by Impala. On the other hand, for polyline intersection based spatial joins, the datasets of both sides in a spatial join are likely to be large which requires spatial partitioning. Partitioning polyline datasets for spatial joins is conceptually more complex than partitioning point datasets as polylines have non-zero spatial extents. Different from HadoopGIS and SpatialHadoop that naturally support partition-based spatial join based on MapReduce and Hadoop, supporting partition-based spatial joins based on Impala is nontrivial, despite several efforts were attempted. Fortunately, our in-house developed LDE engine, which is conceptually equivalent to Hadoop runtime, allows an easier extension to support partition-based spatial joins. We next introduce partition-based spatial joins in Section 3 before we present the proposed polyline intersection based spatial join technique on GPUs and its integration with LDE in Section 4.

3. PARTITION-BASED DISTRIBUTED SPATIAL JOIN

To successfully join large-scale geospatial datasets, especially when the volumes of input datasets or intermediate data exceed the capacity of a single machine, efficient distributed spatial join techniques that are capable of effectively utilizing aggregated resources across multiple computing nodes are essential. When both datasets in a spatial join are large in volumes, a common practice is to spatially partition both input datasets, which can be any spatial data types, for parallel and/or distributed spatial joins on the partitions.

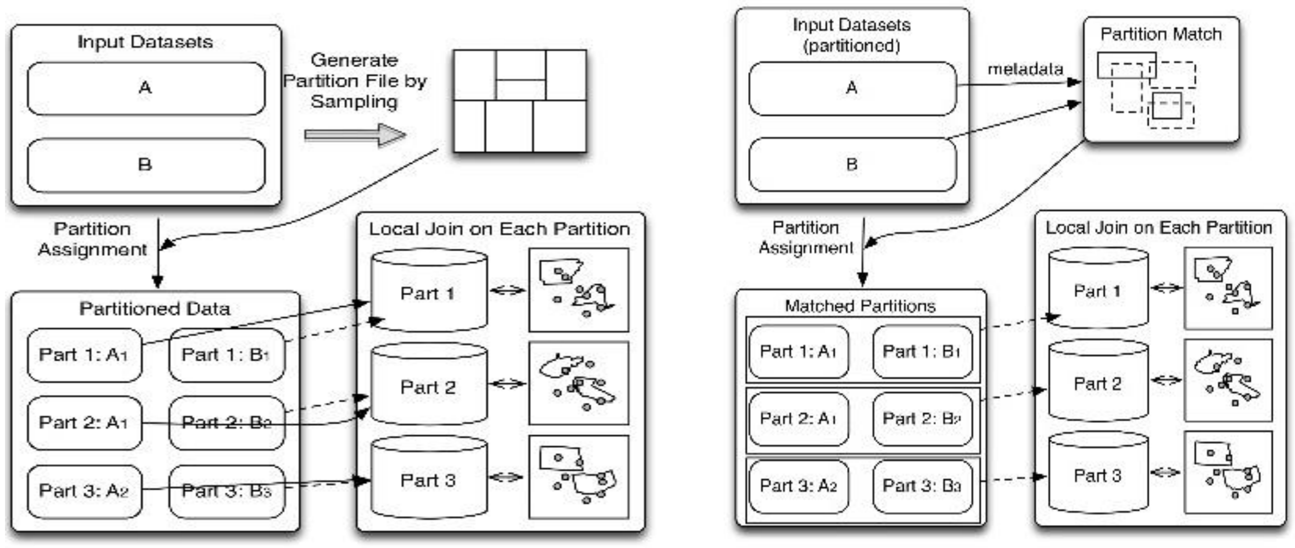


Figure 1 Partition-based Spatial Join

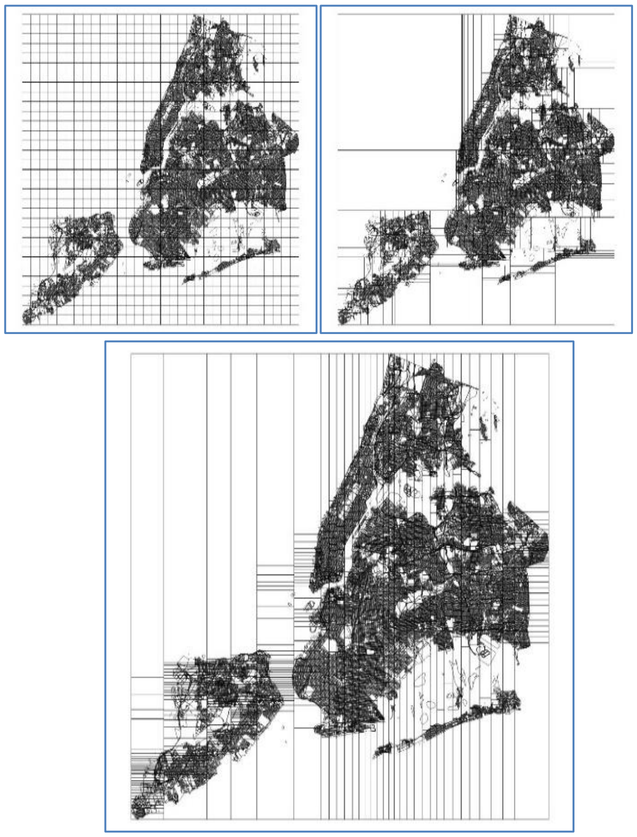
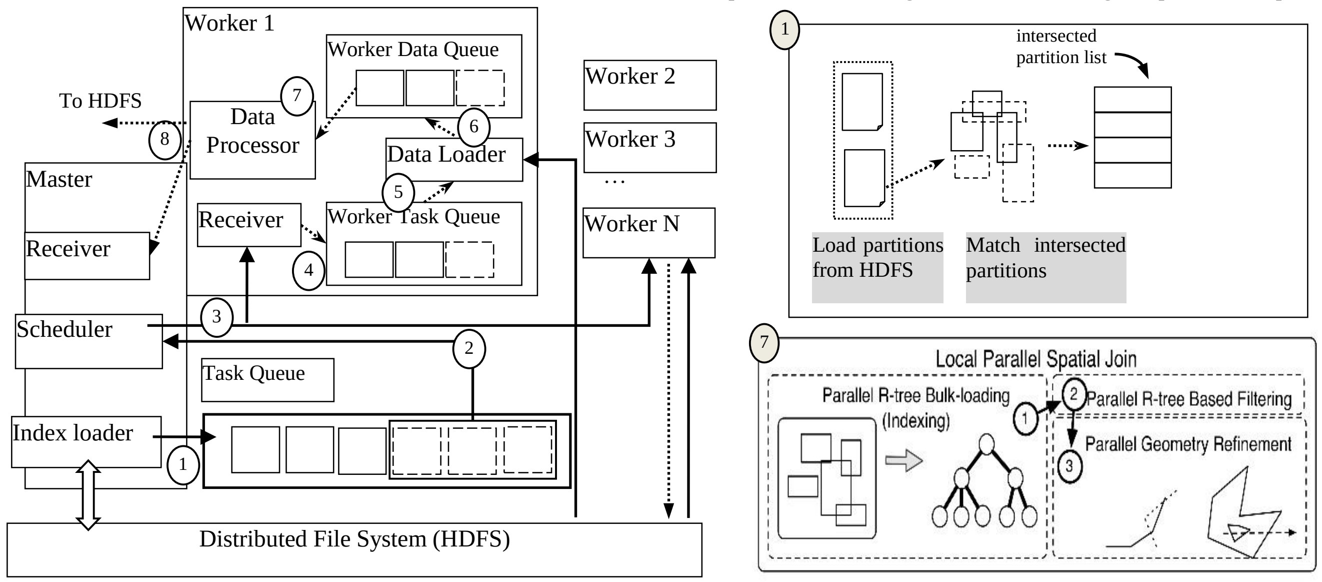


Figure 2 Spatial Partitions of NYC Census Block Data using FGP (top-left), BSP (top-right) and STP (bottom)

In partition-based spatial join, if neither side is indexed, an on-demand partition schema can be created and both sides are partitioned on-the-fly. This approach has been used in previous works [1] [2] [3]. On the other hand, when both datasets have already been partitioned according to a specific partition schema, partitions can be matched as pairs and each pair can be processed independently [2]. The quality of partitions has significant impact on the efficiency of parallel spatial join technique on multiple computing nodes. First of all, a high quality spatial partition schema minimizes expensive intra-node communication cost. Second, such a schema can also minimize the effects from stragglers (slow nodes), which typically dominate end-to-end performance. Therefore, the basic idea of our parallel spatial join technique is to divide the spatial join task into small (nearly) equal-sized and independent subtasks and process those small tasks in parallel efficiently. The left side of Fig. 1 illustrates on-demand partition and the right side of the figure shows the process of matching already partitioned datasets. First, the two input datasets (A and B) are matched as partition pairs using either on-demand partition schema (left) or matching existing partitions (right). Here A and B are represented as partitions where each partition contains a subset of the original dataset, e.g., A_1 and B_1 are in partition 1. Subsequently, partition pairs are assigned to computing nodes and each node performs local spatial join processing.

Several spatial partition schemas have been proposed and implemented previously [16] [17], such as fixed-grid partition (FGP), Binary Split Partition (BSP) and Sort-Tile Partition (STP). Figure 2 shows spatial partition results based on FGP (top-left), BSP (top-right) and STP (bottom) of the 2010 Census Block data in the New York City (NYC), respectively. Although we have empirically chosen STP for spatial partition in this study as the resulting partitions typically have better load balancing features, we refer to [16] [17] for brief introductions to other spatial partition schemas. We note that our system can accommodate all spatial partition schemas as long as spatial data items in a partitioning dataset are assigned to partitions in a *mutually exclusive and collectively exhaustive* manner. In STP, data is first sorted along one dimension and split into equal-sized strips. Within each strip, final partitions are generated by sorting and splitting data according to the other dimension. The parameters for STP are the number of splits at each dimension as well sampling ratio. Different from BSP, STP sorts data at most twice and does not need recursive decompositions, which is more efficient for large datasets based on our experiences. We also note that our decision on adopting STP as the primary spatial partition schema agrees with the finding in SpatialHadoop [2].



Worker 1
Worker Data Queue
To HDFS
7
Data Processor
6
8
Data Loader
Master
5
Receiver
Worker Task Queue
Receiver
4
Worker 2
Worker 3
…
Worker N
Scheduler
3
2
Task Queue
Index loader
1
Distributed File System (HDFS)
1
intersected partition list
Load partitions from HDFS
Match intersected partitions
7
Local Parallel Spatial Join
Parallel R-tree Bulk-loading (Indexing)
1
2
Parallel R-tree Based Filtering
Parallel Geometry Refinement
3

Figure 3 System Architecture and Modules

4. SYSTEM ARCHITECTURE AND IMPLEMENTATIONS

The distributed partition-based spatial join technique is developed by significantly extending the LDE engine we have developed previously. While designed to be lightweight, LDE supports asynchronous data transfer over network, asynchronous disk I/O and asynchronous computing and we refer to [11] for details. The asynchronous design makes the join processing in a non-blocking manner, which can deliver high performance by hiding latency from disk access and network communication. The LDE architecture is extended in several aspects to accommodate partition-based spatial joins. The overall system architecture is illustrated in the left side of Figure 3. First, the master node reads the partition information from HDFS and populates the task schedule queue associated with the scheduler. This is different from the original LDE design for broadcast-based spatial joins where the queue task is populated by sequence-based partitions that are computed on-the-fly. Second, batches are now computed from partition pairs, instead of from partitions of the left side input (point dataset). Subsequently, an optimization technique has been developed to minimize duplicated HDFS disk I/Os within batches. We note that a batch is the basic unit for distributed processing in LDE. The details of task scheduling, batch dispatching and distributed processing are provided in Section 4.1. Second, different from previous studies that rely on open source geometry libraries to perform spatial refinement (polyline-intersection in particular in this study) which only work on CPUs in a serial computing setting, we have developed data parallel algorithms for the spatial refinement that are efficient on both GPUs and multi-core CPUs. The details of the key technical contribution in this study are provided in Section 4.2. We note that, similar to SpatialHadoop, currently spatial partitioning of input datasets is treated as a preprocessing step and therefore it is not shown in Figure 3. By following a few simple naming and formatting conventions, the partitions can be computed either serially, in parallel or imported from third party packages such as SpatialHadoop, which make the architecture flexible.

4.1 Scheduling, Dispatching and Distributed Processing

Step 1 in Figure 3 pairs partitions of both sides of inputs to compute partition pairs. As the numbers of spatial partitions of the two input datasets in a join are typically small, pairing the partitions based on their MBRs is fast using any in-memory algorithms on a single computing node. We thus implement the step at the master node. Assuming both datasets are partitioned and partition boundaries are stored in HDFS, the index loader of the master node in LDE loads partition MBRs of both datasets from HDFS before performing in-memory parallel pair matching to generate a partition pair list in Step 1 as shown in the lower-left side of Figure 3. The details of Step 1 are further illustrated in the top-right part of Figure 3. The matched partition pairs are then pushed into a task queue which will be dispatched by the scheduler at the master node in Step 2. The minimum unit in a batch for the scheduler is a single partition pair. While it is intuitive to use a pair as a batch for distributed execution in a worker node, such design will generate a large number of batches that require substantial communication overheads, which will negatively impact system performance. In addition, such a na ïve scheduling policy will also result in hardware underutilization on worker nodes, especially for GPUs. Unless a matched partition pair involves large numbers of data items in the input datasets, the workload for processing a single pair is unlikely to saturate GPUs that have large numbers of processing units.

In order to reduce network bandwidth pressure and minimize overhead of hardware accelerators, we divide the list of matched pairs into multiple batches where each batch contains multiple partition pairs which are processed one at a time on a worker node. The number of partition pairs in a batch is determined by the processing capability of the worker node, e.g., memory capacity and available processing units. When a worker node joins the master node during the system initialization, the processing capability information is sent to the master node. Based on such information, the size of a batch can be determined. Clearly the batch size for a worker node varies based on the capability of the worker node in our design. Unlike distributed joins in SpatialHadoop where each partition pair is processed by a Mapper in a MapReduce job and requires Hadoop runtime to do the scheduling which is unaware of spatial data, our system is spatial-conscious.

Workload dispatching is achieved by maintaining a receiver thread in each worker node (Step 3). The receiver thread listens to a socket port dedicated for the LDE framework. Note that in each batch, only data locations and offsets to the partitions are stored and transferred between the master node and a worker node. This is because all worker nodes are able to access data directly through HDFS and the data accesses are likely to be local due to HDFS replications. As such, the network communication overhead of transferring batches is very low in our system. Received batches are pushed into a task queue of the worker node (Step 4). A separate thread of the worker node is designated to load data from HDFS for each batch at the worker node (Step 5). The data loaded for each partition pair is kept in an in-memory data queue which will be processed next (Step 6). Since a partition from one dataset may overlap with multiple partitions from the other datasets, there will be duplicated IO requests for the same partition. To reduce the redundant IO requests, we sort the partition pairs in the master node before they are formed as batches. The duplicated partitions will then appear consecutively in the partition pair list. As a result, duplicated partitions for a batch can be detected and only one copy of the duplicated partitions is loaded from HDFS at a worker node. This improvement significantly reduces expensive data accesses to HDFS. The local spatial join logic is fully implemented in Step 7 and the join query results are written to HDFS in Step 8. While we refer to the left side of Figure 3 for the workflow of Steps 1-8, the details of Step 7 are further illustrated in the lower-right part of Figure 3 and will be explained in details next.

4.2 Data Parallel Local Spatial Join on GPUs

As introduced previously, each worker node has a local parallel spatial join module to process partition pairs in a batch. In addition to the designs and implementations for point-in-polygon test based spatial joins and point-to-polyline distance based spatial joins that we have reported previously, in this study, we aim at designing and implementing a new spatial predicate for polyline intersection test that can be integrated into the classic *filter-and-refinement* spatial join framework [12] on GPUs. We reuse our GPU-based R-tree technique [18] for spatial filtering and we next focus on the polyline intersection test spatial predicate.

Assuming the R-Tree based spatial filtering generates a list of polyline pairs where the MBRs of the polylines in each pair intersect. Given two polylines, $P(p_1, p_2, \ldots, p_m)$ and $Q(q_1, q_2, \ldots, q_n)$, where p and q are line segments, we can perform intersection test on P and Q by checking whether any of two line segments in P and Q intersect. As we have a list of candidate pairs, it is intuitive to assign each pair to a processing unit for parallel processing. However, the numbers of line segments of polylines can be very different which may result in poor performance when the na ïve parallelization approach is applied to GPUs due to unbalanced workloads across GPU threads. Perhaps more importantly, as each GPU thread needs to loop through all line segments of another polyline in a matched pair separately in the na ïve parallelization, neighboring GPU threads are not likely to access the same neighboring memory locations. The non-coalesced memory accesses may lower the GPU performance significantly as accesses to GPU memory can be more than order of magnitude slower than accesses to CPU memory, given the quite different cache configurations on typical GPUs and CPUs.

```
Input: polyline representations, candidate pairs
Output: intersection status
Polyline_Intersection:
1: (pid, qid) = get_polyline_pair(block_id)
2: (p_start, p_end) = linestring_offset(pid)
3: (q_start, q_end) = linestring_offset(qid)
4: __shared__ intersected = False
5: for p_linestring from p_start to p_end
6:     for q_linestring from q_start to q_end
7:         __syncthreads()
8:         workload = len(p_linestring) * len(q_linestring)
9:         processed  = 0
10:        while (!intersected && processed < workload)
11:            if (thread_id + processed >= workload) continue
12:            p_seg = get_segment(p_linestring, thread_id)
13:            q_seg = get_segment(q_linestring, thread_id)
14:            is_intersect = segment_intersect(p_seg, q_seg)
15:            if (is_intersect) intersected = True
16:            processed += num_threads_per_block
17:            __syncthreads()
18:        end while
19:        if (intersected)
20:            results[block_id] = True
21:            return
22:    end for //p_linestring
23: end for //q_linestring
```

Figure 4 GPU Kernel for Polyline Intersection Test

As balanced workload and coalesced global memory access are crucial in exploiting the parallel computing power on GPUs, we have developed an efficient data-parallel design on GPUs to achieve high performance. First, we minimize unbalanced workload by applying parallelization at line segment level rather than at polyline level. Second, we maximize coalesced global memory accesses by laying out line segments from the same polyline consecutively on GPU memory and letting each GPU thread process a line segment. Figure 4 lists the kernel of the data parallel design of polyline intersection test on GPUs and more details are explained below.

In our design, each pair of polyline intersection test is assigned to a GPU computing block to utilize GPU hardware scheduling capability to avoid unbalanced workload created by variable polyline sizes. Within a computing block, all threads are used to check line segments intersection in parallel. Since each thread performs intersection test on two line segments where each segment has exactly two endpoints, the workload within a computing block is perfectly balanced. The actual implementation for real polyline data is a little more complex as a polyline may contain multiple *linestrings* which may not be continuous in polyline vertex array. The problem can be solved by recording the offsets of the *linestrings* in the vertex array of a polyline and using the offsets to locate vertices of line segments of the *linestrings* to be tested.

Line 1-3 of the kernel in Figure 4 retrieve the positions of non-continuous *linestrings*, which are iterated in Lines 5 and 6. For each pair of *linestrings*, all threads of a block retrieve line segments in pairs and test for intersection (Lines 10-17). We designate a shared variable, *intersected*, to indicate whether there is any pair of line segments intersected for the polyline pair. Once a segment pair intersects, the *intersected* variable is set to true and becomes visible to all threads within the thread block. The whole thread block then immediately terminates (Lines 18-20). When the thread block returns, the GPU hardware scheduler can schedule another polyline pair on a new thread block. Since there is no synchronization *among* thread blocks, there will be no penalty even though unbalanced workloads are assigned to blocks. Note that the *syncthreads*() statement at Line 17 is to synchronize threads *within* a single block and the overhead is generally acceptable.

Figure 5 illustrates the design of polyline intersection for both multi-core CPUs and GPUs. After the filter phase, candidate pairs are generated based on MBRs of polylines. As we mentioned previously, a pair of polylines can be assigned either to a CPU thread (multi-core CPU implementation) for iterative processing by looping though line segments or to a GPU thread block (GPU implementation) for parallel processing. While GPUs typically have hardware schedulers to automatically schedule multiple thread blocks on a GPU, explicit parallelization on polylines across multiple CPU cores is needed. While we use OpenMP with dynamic scheduling for the purpose of this study, other parallel libraries on multi-core CPUs, such as Intel Thread Building Block (TBB), may achieve better performance by utilizing more complex load balancing algorithms. In both multi-core CPU and GPU implementations, we have exploited native parallel programming tools to achieve higher performance based on a shared-memory parallel computing model. This is different from executing Mapper functions in Hadoop where each Mapper function is assigned to a CPU core and no resource sharing is allowed among CPU cores.

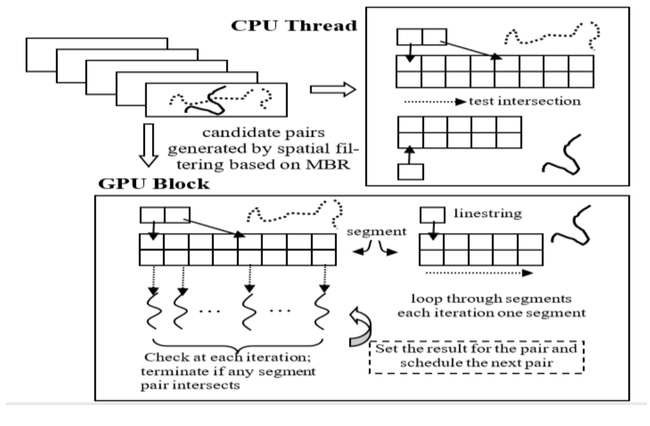

CPU Thread
test intersection
candidate pairs generated by spatial filtering based on MBR
GPU Block
segment
linestring
loop through segments each iteration one segment
Check at each iteration; terminate if any segment pair intersects
Set the result for the pair and schedule the next pair

Figure 5 Data Parallel Polyline Intersection Design

As newer generations of CPUs, such as Intel Haswell and Skylake, that support 256-bit AVX and AVX2 instructions are becoming mainstream, it is also interesting to apply the same data parallel design to multi-core CPUs where each core is enhanced with a Vector Processing Unit (VPU). With a SIMD length of 256/32=8, the VPUs can potentially speed up a CPU core by 8X. Future Intel CPUs will support AVX-512 which brings the SIMD length to 16. Conceptually a GPU streaming multiprocessor is similar to a CPU core with VPU enabled and a CPU thread is functionally equivalent to a GPU thread block. Although physical VPU SIMD length is smaller than GPU computing block size, a SIMD width of 16 is close to GPU warp size (currently 32). Note that a warp is the minimum unit within a thread block to be executed on a multiprocessor in a GPU. We plan to exploit CPU SIMD computing power for polyline intersection test by leveraging our previous experiences on simplification of point-in-polygon test and point-to-polyline distance computation as reported in [19].

5. EXPERIMENTS AND RESULTS

5.1 Experiment Setup

In order to conduct performance study, we have prepared real world datasets for experiments. They are publically accessible to facilitate independent evaluations on different parallel and/or distributed platforms. The experiment is designed to evaluate polyline intersection based spatial joins using two datasets provided by SpatialHadoop[vii], namely TIGER *edge* and USGS *linearwater*. For both datasets, only geometries are used from the original datasets for the experiment purpose and the specifications are listed in Table 1. We also apply appropriate preprocessing on the datasets for running on different systems. For SpatialHadoop, we use the provided R-tree indexing module and leave other parameters by default. For our system, all the datasets are partitioned using Sort-Tile partition (256 tiles for both *edge* and *linearwater*) for partitioned-based spatial joins.

Table 1 Experiment Dataset Sizes and Volumes

Dataset	# of Records	Size
Linearwater	5,857,442	8.4GB
Edge	72,729,686	23.8GB

We have prepared several hardware configurations for experiment purposes. The first configuration is a single node cluster with a workstation that has dual 8 core CPUs at 2.6 GHz (16 physical cores in total) and 128 GB memory. The large memory capacity makes it possible to experiment spatial joins that require a significant amount of memory. The workstation is also equipped with an Nvidia GTX Titan GPU with 2,688 cores and 6 GB memory. Another configuration, which is designed for scalability test, is a 10-node Amazon EC2 cluster, in which each node is a *g2.2xlarge* instance consisting of 8 vCPUs and 15 GB memory. Each EC2 instance has an Nvidia GPU with 1,568 cores and 4 GB memory. We vary the number of nodes for scalability test and term the configurations as EC2-X where X denotes the number of nodes in the cluster. Both clusters are installed with Cloudera CDH-5.2.0 to run SpatialHadoop (version 2.3).

5.2 Results of Polyline Intersection Performance on Standalone Machines

We first evaluate our polyline intersection designs using *edge* and *linearwater* datasets on both multi-core CPUs and GPUs on our workstation and a *g2.2xlarge* instance without involving distributed computing infrastructure. As the polyline intersection time dominates the end-to-end time in this experiment, the performance can be used to evaluate the efficiency of the proposed polygon intersection technique on both multi-core CPUs and GPUs. The results are plotted in Figure 6, where *CPU-Thread* and *GPU-Block* refer to the implementations of the proposed design, i.e., assigning a matched polyline pair to a CPU thread and a GPU computing block, respectively. Note the data transfer time between CPUs and GPUs is included when reporting GPU performance.

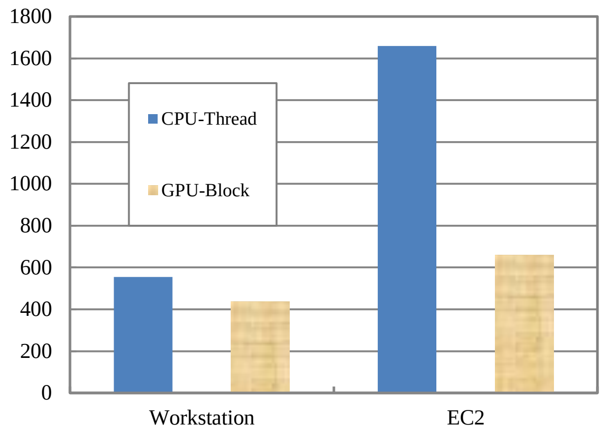


Figure 6 Polyline Intersection Performance (in seconds) on a Standalone Workstation and an EC2 Instance

For GPU-Block, the approximately 50% higher performance on the workstation than on the single EC2 instance shown in Figure 6 represents a combined effect of about 75% more GPU cores and comparable memory bandwidth when comparing the GPU on the workstation and the EC2 instance. For CPU-Thread, the 2.4X better performance on the workstation than that on the EC2 instance reflects the facts that the workstation has 16 CPU cores while the EC2 instance has 8 virtualized CPUs, in addition to Cloud virtualization overheads. While the GPU is only able to achieve 20% higher performance than CPUs on our high-end workstation, the results show 2.6X speedup on the EC2 instance where both the CPU the GPU are less powerful. The reported low GPU vs. CPU speedup on the workstation may actually represent the high efficiency of our polyline intersection test technique on both CPUs and GPUs. Although it is not our intension to compare our data parallel polyline intersection test implementations with those that have been implemented in GEOS and JTS, we have observed orders of magnitude of speedups. As reported in the next subsection, the high efficiency of the geometry library actually is the key source for our system to significantly outperform SpatialHadoop for partition-based spatial joins where SpatialHadoop uses JTS for geometry library.

5.3 Results of Distributed Partition-Based Spatial Joins

The end-to-end runtimes (in seconds) for the experiments *edge-linearwater* experiment under the four configurations (WS, EC2-10, EC2-8 and EC2-6) on the three systems (SpatialHadoop, LDE-MC+ and LDE-GPU) are listed in Table 2. The workstation (denoted as WS) here is configured as a single-node cluster and is subjected to distributed infrastructure overheads. LDE-MC+ and LDE-GPU denote the proposed distributed computing system using multi-core CPUs and GPUs, respectively. The runtimes of the three systems include spatial join times only and the indexing times for the two input datasets are excluded.

From Table 2 we can see that, comparing with SpatialHadoop, the LDE implementations on both multi-core CPUs and GPUs are at least an order of magnitude faster for all configuration. More specifically, LDE-GPU is 22.6X faster on the workstation and 40X-51X faster on EC2 clusters. Similarly, LDE-MC+ is

17.8X faster on the workstation and 17.7X-21.6X faster on EC2 clusters.

The efficiency is due to several factors. First, the specialized LDE framework is a C++ based implementation which can be more efficient than general purpose JVM based frameworks such as Hadoop (on which SpatialHadoop is based). The in-memory processing of LDE is also an import factor as Hadoop is mainly a disk-based system. With in-memory processing, intermediate results do not need to be written to external disks, which is very expensive. Second, as mentioned earlier, the dedicated local parallel spatial join module can fully exploit parallel and SIMD computing power within a single computing node. Our data-parallel designs in the module, including both spatial filter and refinement steps, can effective utilize the current generation of hardware, including multi-core CPUs and GPUs.

From a scalability perspective, the LDE engine has achieved reasonable scalability. When the number of EC2 instances is increased from 6 to 10 (1.67X), the speedups are 1.39X and1.64X for LDE-MC+ and LDE-GPU, respectively. The GPU implementations can further achieve 2-3X speedups over the multi-core CPU implementations, which is desirable for clusters equipped with low profile CPUs while supporting distributed IO.

Table 2 Polyline Intersection based Spatial Join Runtimes (in seconds)

	WS	**EC2-10**	**EC2-8**	**EC2-6**
SpatialHadoop	9887	3886	5613	6915
LDE-MC+	554	219	260	360
LDE-GPU	437	97	114	135

6. CONCLUSION AND FUTURE WORK

In this study, we have designed and implemented polyline intersection based spatial joins on GPU-accelerated clusters. By integrating distributed processing and parallel spatial join techniques on GPUs within a single node, our proposed system can perform large-scale spatial joins effectively and achieve much higher performance than the state-of-the-art systems. As for future work, we plan to further improve the single node local parallel spatial module by adding more spatial operators with efficient data-parallel designs. We also plan to develop a scheduling optimizer for the system that can perform selectivity estimation to help dynamic scheduling to achieve higher performance.

Acknowledgement: this work is partially supported by NSF Grants IIS-1302423 and IIS-1302439.

7. REFERENCE

1. A. Aji, F. Wang, et al (2013). Hadoop-gis: A high performance spatial data warehousing system over mapreduce. In *VLDB, 6(11)*, pages 1009–1020.
2. E. Eldawy and M. F. Mokbel (2015). SpatialHadoop: A MapReduce Framework for Spatial Data. In Proc. IEEE ICDE'15.
3. S. You, J. Zhang and L. Gruenwald (2015). Large-Scale Spatial Join Query Processing in Cloud. In Proc. IEEE CloudDM'15.
4. J. Yu, J. Wu and M. Sarwat (2015). GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In Proc. ACM-GIS.
5. D. Xie, F. Li, B. Yao, G. Li, L. Zhou and M. Guo (2016). Simba: Efficient In-Memory Spatial Analytics. In Proc. SIGMOD.
6. S. You, J. Zhang and L. Gruenwald (2015). Scalable and Efficient Spatial Data Management on Multi-Core CPU and GPU Clusters: A Preliminary Implementation based on Impala, in Proc. IEEE HardBD'15.
7. M. Zaharia, M. Chowdhury et al (2010). Spark: Cluster Computing with Working Sets. In Proc. HotCloud.
8. M. Kornacker and et al. (2015). Impala: A modern, open-source sql engine for hadoop. In Proc. *CIDR*'15.
9. K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang and J. H. Saltz (2012). Accelerating pathology image data. Proc. VLDB Endow., vol. 5, no. 11, p. 1543–1554.
10. J. Zhang, S. You and L. Gruenwald (2014). Large-Scale Spatial Data Processing on GPUs and GPU-Accelerated Clusters. ACM SIGSPATIAL Special, vol. 6, no. 3, pp. 27-34.
11. J. Zhang, S. You and L. Gruenwald (2015). A Lightweight Distributed Execution Engine for Large-Scale Spatial Join Query Processing. In Proc. IEEE Big Data Congress'15.
12. E. H. Jacox and H. Samet (2007). Spatial Join Techniques. ACM Trans. Database Syst., vol. 32, no. 1, p. Article #7.
13. S. You, J. Zhang and L. Gruenwald (2015). You, J. Zhang and L. Gruenwald (2015). Spatial Join Query Processing in Cloud: Analyzing Design Choices and Performance Comparisons. In Proc. IEEE HPC4BD.
14. A. Aji, G. Teodoro and F. Wang (2014). Haggis: turbo-charge a MapReduce based spatial data warehousing system with GPU engine. In Proc. ACM BigSpatial'14.
15. H. Chavan, R. Alghamdi and M. F. Mokbel (2016). Towards a GPU Accelerated Spatial Computing Framework. In Proc. IEEE HardBD.
16. H. Vo, A. Aji and F. Wang (2014). SATO: a spatial data partitioning framework for scalable query processing. In Proc. ACMGIS'14.
17. A. Eldawy, L. Alarabi and M. F. Mokbel (2015). Spatial Partitioning Techniques in SpatialHadoop. Proc. VLDB Endow., vol. 8, no. 12, p. 1602-1605.
18. J. Zhang and S. You (2012). Parallel spatial query processing on GPUs using R-trees. In Proc. ACM BigSpatial, 23-32.
19. J. Zhang and S. You (2014). Large-Scale Geospatial Processing on Multi-Core and Many-Core Processors: Evaluations on CPUs, GPUs and MICs. CoRR abs/1403.0802

[i] http://aws.amazon.com/ec2/instance-types/
[ii] http://hadoop.apache.org/docs/r1.2.1/streaming.html
[iii] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[iv] https://www.threadingbuildingblocks.org/
[v] http://www.nvidia.com/object/cuda_home_new.html
[vi] https://en.wikipedia.org/wiki/Well-known_text
[vii] http://spatialhadoop.cs.umn.edu/datasets.html