Scalable and Efficient Spatial Data Management on Multi-Core CPU and GPU Clusters: A Preliminary Implementation based on Impala

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

Jianting Zhang
Department of Computer Science
The City College of New York
New York, NY, USA
jzhang@cs.ccny.cuny.edu

Le Gruenwald
Dept. of Computer Science
The University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

***Abstract*— Fast increasing volumes of spatial data has made it imperative to develop both scalable and efficient spatial data management techniques by leveraging modern parallel hardware and distributed systems. By integrating a leading open source Big Data system called Impala and our previous work on data parallel designs for spatial indexing and query processing, we have developed ISP-MC+ and ISP-GPU for large-scale spatial data management on computer clusters equipped with multi-core CPUs and Graphics Processing Units (GPUs), respectively. Both ISP-MC+ and ISP-GPU have shown high efficiency and good scalability on a 10-node Amazon EC2 cluster equipped with multi-core CPUs and GPUs. Comparison with a baseline implementation using traditional techniques on a single CPU core have demonstrated orders of magnitude of speedups on a real world dataset with hundreds of millions of point locations.**

Keywords*— *Spatial Data, Multi-Core CPU, GPU, Impala, System Integration, High-Performance

I. INTRODUCTION

It has been estimated that 80% of information over the Internet has a spatial component. The increasingly popular mobile devices have generated tremendous amounts of point data, such as GPS, cellular and Wifi locations. Advanced environmental sensing technologies and scientific simulations have also generated large amounts of geo-referenced spatial data. Efficiently managing big spatial data in a scalable way is technically challenging. Existing techniques either focus on scalability (e.g., HadoopGIS [1] and SpatialHadoop [2] based on Hadoop/MapReduce) or single-node efficiency (e.g., in-memory R-Tree indexing [3] and TOUCH based spatial joins [4]). To achieve both efficiency and scalability for large-scale spatial data processing, we have designed and implemented a prototype system called In-memory Spatial Processing (ISP) by significantly extending Impala [5], a leading open source Massively Parallel Processing (MPP) SQL engine for relational data, to support spatial data types and spatial operations. ISP has two variations: ISP-MC+ for multi-core CPU clusters and ISP-GPU for GPU equipped clusters. ISP-MC+ and ISP-GPU share the data parallel designs and implementations of spatial data management techniques that were initially designed for single-node GPUs as reported in our previous works [6,7]. Most of ISP modules are developed on top of the Thrust parallel library [8] that comes with CUDA SDK which allows us to compile the same set of implementations for both multi-core CPUs and GPUs [6]. As such, we will use ISP to refer to the generic designs and implementations and only distinguish ISP-MC+ and ISP-GPU in experiments.

ISP extends Impala's frontend to parse spatial SQL syntax and generate logical and physical query plans. ISP also reuses scheduling, data communication and parallel disk I/O modules provided by Impala backend for distributed computing. At the individual computing nodes, ISP provides on-the-fly spatial indexing and spatial joins on multi-core CPUs and GPUs using native parallel programming tools to achieve high performance. Our experiments on a real-world large dataset have shown orders of magnitude of speedups when compared with a baseline implementation using traditional techniques on a single CPU core. Experiments on a 10-node EC2 Cloud cluster have also demonstrated good scalability. The results clearly suggest that query performance can be significantly improved through a combination of hardware acceleration, in-memory processing, architectural evolution of big data system infrastructure as well as new parallelization-friendly domain-specific data structures. As Impala is designed to support relational data and our extension focuses on spatial data, we believe the experiences learnt in this study can be valuable to the future designs of extensible big data systems that support semi-structured data natively, in a way similar to object-relational extensions to relational databases in the past.

For the rest of the paper, we first introduce background and related work, including a brief introduction to Impala, in Section II. Section III presents the system architecture of ISP and its implementation details. Section IV provides experiment results on large-scale point-in-polygon test based spatial join query processing using two real point datasets: ~170 million taxi trip pickup locations in the New York City (NYC) and ~375 million global species occurrence records. Finally Section V is the conclusion and future work directions.

II. BACKGROUND AND RELATED WORK

Existing big data technologies such as MapReduce/Hadoop have been popular for distributed data storage and processing over the past few years. In order to process big spatial data efficiently, several extensions have been developed to support spatial data processing on Hadoop, including Hadoop-GIS [1], SpatialHadoop [2] and ESRI GIS Tools for Hadoop [9]. As memory is getting significantly cheaper and computers are increasingly equipped with large memory capacities, there are considerable research and application interests in processing large-scale data in memory to reduce disk I/O bottlenecks and achieve better performance. Newer generations of big data systems such as Apache Spark [10] and Cloudera Impala [5], although still support HDFS (Hadoop Distributed File System [11]) for persistent storage, are able to take advantage of the

large memory capacities and achieve higher performance. However, we are not aware of existing systems that use or extend in-memory big data systems for large-scale spatial data processing, especially for complex spatial joins. In-memory spatial processing techniques, such as in-memory R-Tree indexing for speeding up topological relationship test [3] and the in-memory data structure in TOUCH for efficient spatial join [4] have been recently proposed. Our previous work on grid-file and R-Tree based spatial indexing and query processing on GPUs can also be considered as in-memory techniques [6,7], where GPU-specific characteristics, such as data parallelism, control divergence and coalesced memory accesses, are carefully considered. However, these techniques are designed for single-node computation and cannot easily scale out to distributed machines to process larger scale data.

Traditional MapReduce jobs usually incur large amounts of disk I/Os to store intermediate results. Cloudera Impala [5], as a new generation big data system, is designed to take advantage of large memory capacities to significantly reduce disk I/O overheads during data processing. As a component in the Hadoop ecosystem, Impala supports HDFS and can read/write data in different formats. Impala has a frontend to parse SQL statements and generate logical query plans with rule-based optimizations. By using metadata and hardware configuration information, Impala frontend is able to generate efficient physical query plans by incorporating cost-based optimizations. A query plan is then sent to an Impala backend instance that works as the master. The master coordinates with a set of worker Impala instances to process a query in a distributed manner. The query results are gathered by the master and sent back to clients (e.g., a shell program that initiates the query). Different from traditional MapReduce/Hadoop based systems that write intermediate query results to HDFS, an Impala instance processes all the computation that is being assigned to it completely in memory without touching HDFS except for explicit outputting.

As one of the few open source big data systems with a C/C++ based execution engine, Impala is ideal to serve as the base for further extensions when performance is critical. In particular, as currently Java does not support exploiting SIMD (Single Instruction Multiple Data) computing power [12] on either CPUs or GPUs [13], C/C++ language interfaces might be the most viable option to effectively utilize hardware accelerations. While ISP-GPU currently focuses on in-memory spatial query processing using GPUs, we also plan to exploit SIMD computing power for high performance on Vector Processing Units (VPUs) that come with commodity multi-core CPUs [12]. Some preliminary results on comparing the SIMD computing on GPUs, multi-core CPUs and Intel Xeon Phi accelerators for point-to-polyline distance computation and point-in-polygon tests have been reported in [14].

To the best of our knowledge, currently Impala is the only open source big data system that supports Just-In-Time (JIT) compilation using the leading open source LLVM compiler [15]. JIT is essential for high-performance query processing on modern hardware [5] by supporting various types of dynamic code optimizations, such as compiling complex expressions and seamlessly integrating User Defined Functions (UDFs) with native machine code. LLVM based JIT for multiple hardware architectures also makes it possible to dynamically dispatch native code to both multi-core CPUs and GPUs for higher performance in our future work.

III. ISP System Architecture and Implementations

In order to extend Impala for high-performance spatial query processing on multi-core CPUs and GPUs, we have made three major extensions. First, we modify the Abstract Syntax Tree (AST) module of Impala frontend to support spatial query syntax. Second, we represent geometry of spatial datasets as strings to support spatial data accesses in Impala and prepare necessary data structures for GPU-based spatial query processing. Third, we have integrated our single-node GPU-based spatial data management techniques with Impala to support large-scale spatial data processing on GPU-equipped clusters. The GPU code written in CUDA and on top of the Thrust parallel library is compiled to a shared library and is invoked in the spatial data management module implemented as a subclass of *ExecNode* called *SpaitalJoin* in Impala. Similarly multi-core CPU code is developed on top of the Thrust library and Intel TBB library [6]. The architecture of ISP and its components are shown in Fig. 1.

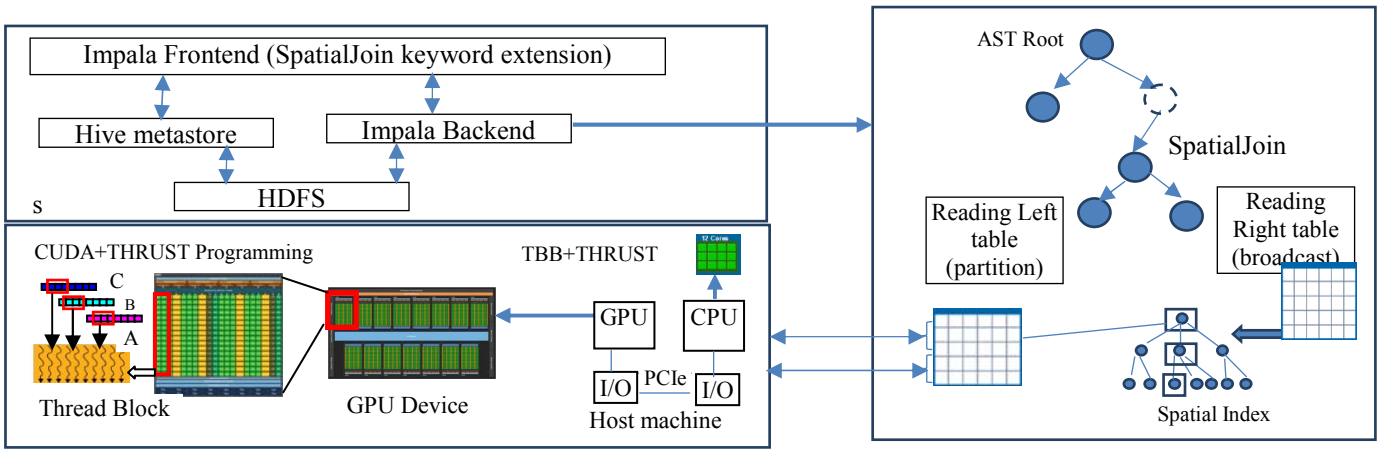

Fig. 1 ISP System Architecture and Components

ISP represents geometry of spatial objects using the Well-Known Text (WKT [16]) format for three reasons. First, the format is simple and is well accepted by the geospatial processing and spatial data management communities. Second (and more importantly), string data type is natively supported by virtually all data management systems including Impala. Using WKT for geometry allows us to support spatial data without significantly changing the underlying big data system infrastructure. Third, as Impala (and hence ISP) is main-memory resident, parsing WKT presentation into in-memory binary representation is one-time cost (per query) and the overhead is typically acceptable, especially when querying complex spatial data that involves expensive floating point computation. As both Impala and ISP are designed for analytical queries (e.g., table scans and aggregations on raw and joined tables), we do not expect outputting a large number of spatial objects where converting binary representation to WKT format for outputting can be another bottleneck. Nevertheless, although currently we consider using WKT for geometry representation to be a justifiable tradeoff between system efficiency and development complexity, we plan to support binary representation of spatial data natively in ISP both in-memory and on disks in our future work.

ISP supports both "single-sided" ad-hoc spatial queries and "double-sided" spatial queries through different mechanisms. For "single-sided" ad-hoc spatial queries, one or more geometric objects are provided in WKT format in a SQL statement. For "double-sided" spatial queries, two tables each with a geometry attribute are involved and are typically referred to as spatial joins (a SQL example is shown at the beginning of Section IV). Although full cross join is embarrassingly parallel and is naturally supported in Impala, the complexly is O(m*n), where m and n are the numbers of tuples in the two tables being joined. A naïve implementation of cross join performs poorly and is impractical for large-scale data, even on GPUs and multi-core CPUs. As such, spatial indexing for at least one side is necessary for "double-sided" spatial queries. ISP-GPU offloads both spatial indexing, spatial filtering and spatial refinements to GPUs to utilize the massively data parallel computing power on GPUs. On the other hand, ISP-MC+ makes full use of multi-core CPUs that support coarse-grained task parallelism better. As shown in section IV, although ISP-MC+ on 16-core Intel CPUs is generally slower than ISP-GPU on an NVIDIA GTX Titan GPU using a single computing node, the performance is comparable which makes integrated CPU-GPU processing interesting. We next briefly introduce the row batch based cross join processing framework in Impala, which is originally designed for relational data on a single CPU core, before presenting our GPU and multi-core CPU extensions for spatial data.

When executing a SQL query on an Impala instance, starting from the root of an AST, SQL clauses that are associated with AST nodes are evaluated top-down using the appropriate data partition(s) that are assigned to the Impala instance on a computing node. When joining two tables, the AST node corresponding to the join has two child nodes with necessary information of the two tables as well as expressions corresponding to a *WHERE* clause. Starting from here, Impala requests batches of tuples (row batches) from the two tables iteratively to process the join in batches and recursively for sub-queries. Depending on whether the data on the right side of a join is partitioned or broadcast, it is possible to retrieve either a partition or the complete right side table. In the current implementation of Impala (as of version 2.0), the left side table is always partitioned for parallel and distributed computation among multiple Impala instances. The design naturally favors asymmetrical joins, i.e., big table (partitioned) on the left side and small table (broadcast) on the right side. We found that the pattern actually captures real world spatial data management tasks very well. For example, in our taxi trip data management applications [6], while the numbers of pickup and drop-off locations increase quickly (half a million a day and ~170 million a year in New York City – NYC), the underlying urban infrastructure data such as road network and census block data are "small" in volume (<200K records). As such, ISP currently focuses on joining "big" point data on the left and "small" infrastructure data on the right, e.g., road network and administrative zones. We plan to support symmetric spatial joins (where both sides can be "big") in our future work.

The implementation of our spatial join extension in Impala on GPUs is as follows. First, we iteratively retrieve the geometry column of tuples of the "small" table and build a spatial index for all the tuples in it. In this study, we apply our previous work on GPU-based R-Tree technique [7] for spatial indexing. We note that retrieving the "small" table from HDFS can be efficiently done using multi-threaded I/O supported by Impala. Second, we iterate through all the row batches of the left side table that are assigned to an Impala instance sequentially to perform the spatial join. For each row batch, we use GPUs to parallelize tuple evaluations. Non-spatial sub-expressions are evaluated first on CPUs before the spatial query is evaluated on GPUs. This is because spatial operations are typically more expensive and can benefit from filtering based on non-spatial criteria, in addition to GPU hardware accelerations of floating point computation. The geometry of a whole row batch of the left side table is transferred to GPUs for parallel query against the spatial index on the GPUs built in the first step. The query result is then transferred back to CPUs in the form of a vector of identifier pairs. Third, tuples of the left side table and right side table are located based on the identifier pairs and they are concatenated (possibly after applying a projection operator) before written to an output tuple buffer to be consumed by the upper level AST nodes for subsequent processing in row batches, e.g., aggregations (at the same level) and upper level SQL clauses (if a sub-query is involved). The process of a point-in-polygon test based spatial join using R-Tree in ISP is illustrated in Fig. 2.

While we refer to our previous works for the details on GPU-based data parallel spatial indexing and query processing on a single node [6, 7], we would like to provide more details on the implementation of the system integration. First, as Structures of Arrays (SoA) are generally more efficient than Arrays of Structures (AoS) on GPUs [17], we store the geometry of both left side and right side tables as binary arrays, including both x/y coordinates and associated auxiliary data for marking the boundaries of complex structures in polygons and polylines. The arrays are filled when WKT strings are parsed

and are subsequently transferred to GPU memory. In addition, as the sizes of the arrays can be derived from the metadata of the inputs, we allocate memory for them only once when they are created. While newer generation GPUs support dynamic memory allocations, the performance is generally better when memory is allocated in big chunks. We have also found a similar pattern on CPUs when compared with using C++ STL classes [6]. Second, we build the spatial index for the right side table in a spatial join on GPUs which will reside in GPU memory until all row batches of the left side table are processed. Reusing the spatial index for the right side table across all the left row batches improves system performance. Given that the number of spatial objects (tuples) in the right side table is typically small and indexing is performed at the Minimum Bounding Rectangle (MBR) level, the memory footprint on GPU is typically acceptable. Third, there is a tradeoff between CPU memory pressure and GPU utilization in setting row batch sizes and related parameters. Impala divides tuples that are assigned to an instance into row batches so that they can be processed in a pipelined manner across multiple AST nodes to lower the memory footprint of intermediate results that need to be materialized. On CPUs, row batch sizes can be set to smaller numbers (e.g., a few thousands) when a small memory footprint is desirable. However, using a small row batch size in ISP-GPU will significantly underutilize GPU massively data parallel processing power and make it difficult to overshadow the data transfer overhead between CPUs and GPUs. While we set row batch sizes empirically in this study (Section IV), we plan to optimize row batch sizes in our future work by exploring the opportunities in both reducing Impala memory footprint on CPUs and using streaming technologies on GPUs [17] to process multiple streams (independent row batches) concurrently. We skip the implementation details of ISP-MC+ as the parallel primitives based implementations on GPUs can be easily ported to multi-core CPUs [6].

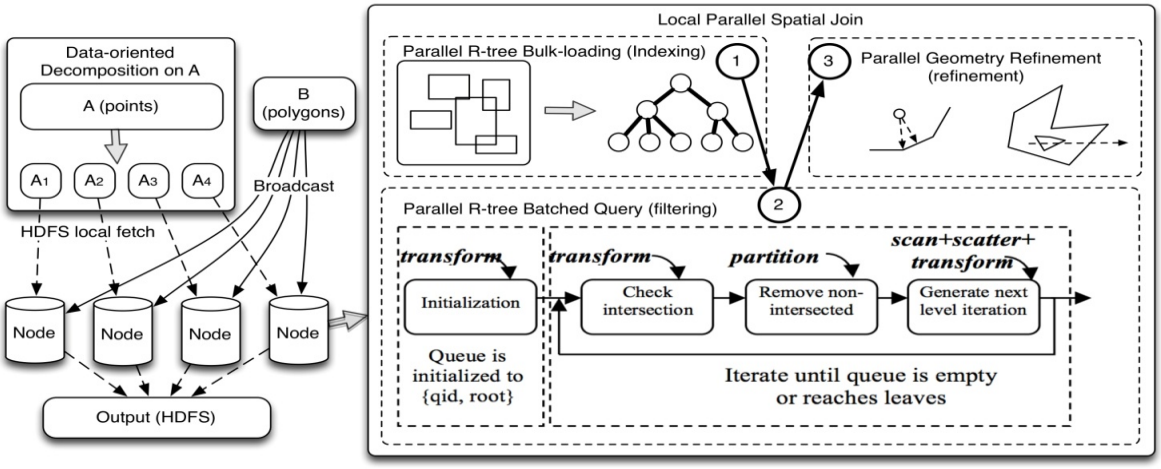

Fig. 2 Illustration of Point-in-Polygon based Spatial Join Processing in ISP-GPU

IV. Performance Evaluation

While ISP supports quite a few frequently used spatial query operations, in this section, we will use point-in-polygon test based "double-sided" spatial join for performance evaluation purposes. Given a point dataset on the left and a polygon dataset on the right, the spatial join between them can be expressed as a SQL statement as following:

```
SELECT point.id, polygon.id
FROM point SPATIAL JOIN polygon
WHERE ST_WITHIN(point.geom, polygon.geom)
```

Here *ST_WITHIN* is a function to test the containment spatial relationship between a point and a polygon. We have used a popular ray-tracing algorithm to implement the point-in-polygon test which has a complexity of O(V), where V is the number of vertices of the polygon. Clearly point-in-polygon test based spatial joins are computationally intensive and are suitable for GPU accelerations.

A. Data and Experiment Setup

In the first experiment, the point dataset is the complete collection of taxi pickup locations in NYC in 2013 ([18], termed as *taxi*) and the polygon dataset is the Census 2010 polygons at the census block level ([19], termed as *nycb*). The number of point data items is ~170 million and the number of polygons is ~40 thousand. Their data volumes in WKT format in HDFS are 6.9 GB and 18.7 MB, respectively. The average number of polygon vertices of *nycb* is 9.

In the second experiment, the point dataset is the geometry of species occurrence records of the Global Biodiversity Information Facility (GBIF) [20] repository (snapshot 08/02/2012, termed as *gbif*) and the polygon dataset is the World Wild Fund (WWF) global ecological region [21] data (termed as *wwf*). The number of point data items is about 375 million and the number of polygons is 14,458. Their data volumes in HDFS are 12.9 GB and 149.8 MB, respectively. The average number of polygon vertices of *wwf* is 279.

Our experiments are performed on both a high-end GPU-equipped workstation and a 10-node Amazon EC2 *g2.2xlarge* GPU cluster. The workstation is equipped with dual 8-core Intel Sandy Bridge 2.6 GHZ CPUs, 128 GB memory, 8 TB HDD and an NVIDIA GTX TITAN GPU. The GTX TITAN GPU has 6 GB GDDR5 memory and 2,668 CUDA cores. All Amazon *g2.2xlarge* instances (computing nodes) are equipped with 8 vCPU (Intel Sandy Bridge 2.6 GHZ), 15 GB memory,

60 GB SSD and an NVIDIA GPU with 4 GB graphics memory and 1,536 CUDA cores. All machines are running CentOS 6.5 and Hadoop 2.5.0 from Cloudera CDH 5.2.0 with default settings. We empirically set the row batch capacity in Impala to accommodate a whole HDFS data block (64 MB by default).

We have designed two groups of experiments to test the efficiency and scalability of ISP. First, we experiment on the single-node performance and system infrastructure overhead (incurred by Impala) on the workstation by comparing with a native implementation using the same designs. Second, we experiment on the scalability of ISP-GPU and ISP-MC+ by using 2-10 Amazon EC2 instances.

B. Single-Node Performance and Infrastructure Overhead

The single-node performance for the two experiments is listed in the first two columns of Table 1. The runtimes are 96 seconds for *taxi-nycb* and 1,822 seconds for *gbif-wwf* for the ISP-GPU implementation. ISP-MC+ performs a little worse than ISP-GPU but still comparable: 130 seconds for *taxi-nycb* and 2,816 seconds for *gbif-wwf*. ISP-GPU is 1.35X (130/96) faster than ISP-MC+ for *taxi-nycb* and 1.55X (2816/1822) faster than ISP-MC+ for *gbif-wwf*. The comparable performance between ISP-GPU and ISP-MC+ is largely due to applying the same set of data parallel deigns and parallel primitives based implementations, which are efficient on not only GPUs but also multi-core CPUs [6].

To appreciate the performance of ISP, a serial implementation [1] using libspatialindex [22] for spatial indexing/filtering and GDAL [23] (which uses GEOS [24] for geometry operations) for spatial refinement can only achieve 138 points per second using a subset of GIBF data with 10 million points on an Intel Xeon processor (2.0 GHZ). In contrast, ISP-GPU has achieved a rate of 206 thousand points per second using a single GPU which amounts to a 1,491X speedup. When comparing ISP-MC+ with the baseline implementation (965X speedup), while the multiple CPU cores and higher CPU frequency may explain up to 21X speedups (16*2.6/2.0), the rest of the speedups are largely due to our data parallel designs and better use of memory capacity and we refer to [6] for more discussions.

TABLE 1 RUNTIMES OF STANDALONE AND ISP ON A SINGLE NODE

	ISP-GPU	**ISP-MC+**	**GPU-Standalone**	**MC-Standalone**
taxi-nycb (s)	96	130	50	89
GBF-WWF(s)	1822	2816	1498	2664

To understand the system infrastructure overheads, we have implemented the standalone versions on both multi-core CPUs and GPUs, which follow the same designs as we have used previously but directly work on raw input data without requiring the Impala infrastructure. The standalone versions avoid several infrastructure overheads, including SQL parsing, multi-node scheduling, HDFS, and perhaps more importantly, data copying to/from row batch structures. The performance of the standalone versions are listed in the last two columns of Table 1. Clearly, the system infrastructure overhead is quite significant for ISP-GPU: almost 50% (46s) in the *taxi-nycb* experiment and 17% (324s) in the *gbif-wwf* experiment. The overheads are 20% and 8.3% for ISP-MC+, respectively. Although still significant, the infrastructure overheads are much smaller for ISP-MC+ than for ISP-GPU in both experiments. As the experiments become more floating point computing intensive where computation becomes dominate, we expect the system infrastructure overheads continue to decline for both ISP-GPU and ISP-MC+.

It is also interesting to look into the distributions of runtimes of ISP-GPU in the two experiments for different stages of the end-to-end spatial join query processing. Impala (and hence ISP-GPU and ISP-MC+) uses multi-threaded asynchronous disk I/Os; there is a significant overlap between disk I/Os and processing which makes it hard to measure I/O times alone. As such, we focus on pre-processing (WKT string parsing), spatial query processing (indexing, filtering and refinement) and post-processing (outputting spatial join results to row batches). In the *taxi-nycb* experiment, the pre-processing and post-processing (6s) are performed on multi-core CPUs. The runtimes for the two stages are 11-12s and 6s for both ISP-GPU and ISP-MC+. The runtimes for spatial query processing for the *taxi-nycb* experiment are 52.7s on GPUs and 97.8s on multi-core CPUs, respectively. Clearly, despite that ISP-GPU performs only 1.35X faster than ISP-MC+ in the *taxi-nycb* experiment overall, the main spatial query processing step is about 1.85X faster. For the *gbif-wwf* experiment, as the pre-processing and post-processing runtimes (~20s in total) are insignificant when compared to spatial query processing runtime, the speedup of the spatial query processing runtime stage (1.58X) is about the same as the overall speedup (1.55X). The speedups are comparable to what we have reported in [14] using the same high-end workstation.

C. Scalability on EC2 Clusters

We conduct scalability tests on Amazon EC2 clusters with up to 10 *g2.2xlarge* instances. As the memory capacity of the instances is 15 GB, we are not able to run the *taxi-nycb* workload with four or fewer nodes. Also due to the memory capacity constraint, we are not able to experiment on the complete WWF dataset on the 10-node cluster. As such, we have extracted a subset of the WWF dataset with approximately 50 million points and we call the subset G50M and label the experiment as *G50M-wwf*. The scalability results for *taxi-nycb* and *G50M-wwf* experiments are plotted in Fig. 3.

For the *taxi-nycb* experiment, as the number of computing nodes increases, the runtime decreases almost linearly which indicates good scalability for both the GPU and the multi-core CPU implementations. For the *G50M-wwf* experiment, the scalability of ISP-GPU is approximately linear until the number of nodes is increased to above 8. Almost no performance gains are observed when the number of instances is increased from 8 to 10. On the other hand, ISP-MC+ scales up to 10 nodes, although the slope is flatter when the number of instances is increased from 6 to 10 than from 2 to 6 (i.e., scalability becomes lower). Overall, there is a 1.76X speedup for ISP-MC+ and 1.56X speedup for ISP-GPU when the number of nodes is increased from 6 to 10 (1.66X) for the *taxi-*

[1] Details available at http://www-cs.ccny.cuny.edu/~jzhang/zs_gbif.html. Code, data and instructions are released to repeat the experiments.

nycb experiment, which is considered as pretty good. In the *G50M-wwf* experiment, the speedups are 3.19X for ISP-MC and 2.57X for ISP-GPU when the number of node is increased from 2 to 10 (5X), which is still decent with respect to parallelization efficiency (defined as the ratio of performance speedup over increase of parallel processing units).

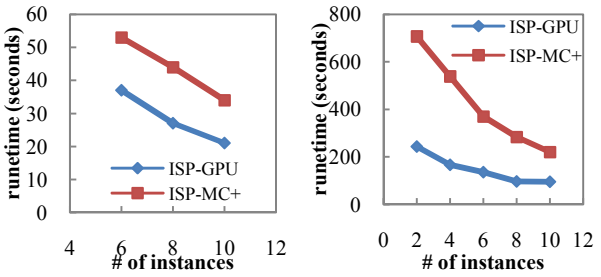

Fig. 3 Scalability Test Results of ISP-GPU and ISP-MC+ for taxi-nycb (left) and G50M-wwf (right) Experiments

The lower speedups when the numbers of computing nodes become higher in the *G50M-wwf* experiment might be largely due to the static scheduling policy imposed by Impala. By examining the G50M point dataset in HDFS, we found that there were 14 HDFS data blocks, which makes the end-to-end runtime about the same using 8-13 computing nodes, as it is determined by the runtime of the computing nodes that process the most (two) blocks. Increasing the number of blocks is likely to reduce load unbalancing to scale further. However, as discussed earlier, as per-node work load decreases, GPUs will likely be underutilized and will negatively hurt the overall performance. The small per-node work load on GPUs is also likely to incur load unbalancing among GPU threads and thread blocks which may further decrease ISP-GPU performance. Since the number of CPU cores is much smaller than the number of GPU cores, the intra-node load unbalancing is less likely to be an issue for ISP-MC, which might explain its better scalability than ISP-GPU in both experiments.

It is interesting to observe that when comparing ISP-GPU with ISP-MC+ on the EC2 cluster, ISP-GPU is 1.43X to 1.63X faster for the *taxi-nycb* experiment and 2.74X to 3.24X faster for the *G50M-wwf* experiment, which are higher than the results on the workstation. This is likely due to the fact that the CPUs equipped with the high-end workstation have 2X cores than those on EC2 nodes while the differences among their GPUs are smaller (1.75X more CUDA cores and 1.5X GPU memory). The results may suggest that GPU acceleration is more profitable for computing nodes with less powerful CPUs.

V. Conclusion and Future Work

We reported our designs and implementations of an in-memory spatial data management system on multi-core CPU and many-core GPU clusters by extending the open source Cloudera Impala big data system for distributed spatial join query processing. Experiments on the initial implementations have revealed both advantages and disadvantages of extending a tightly-coupled big data system to support spatial data types and their operations. Both ISP-MC+ and ISP-GPU have achieved high efficiency by adopting SoA data layout for geometric data, implementing the point-in-polygon test based on SoA, and parallelizing the geometric operation on both multi-core CPUs and GPUs. They have also demonstrated reasonable scalability based on the results using a 10-node EC2 cluster equipped with both multi-core CPUs and GPUs.

For future work, we would like to further improve the performance of the current ISP implementations that have been discussed inline, e.g., optimizing row batch sizes to balance between CPU memory footprint and GPU utilization efficiency, supporting symmetrical spatial joins where both sides require partitions, and integrating GPU and CPU processing for higher performance. Furthermore, we plan to research on a new framework that supports spatial data natively, which might be more extensible and more efficient. As a first step, we are in the process of building a distributed data communication layer that allows easy, efficient and dynamic mapping among data parallel processing primitives and distributed data communication primitives.

ACKNOWLEDGEMENT
This work is supported through NSF Grants IIS-1302423 and IIS-1302439 and PSC-CUNY grant 66724-00 44.

References

[1] A. Aji et al (2013). Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. In Proc. VLDB, 6(11), 1009-1020, 2013.
[2] A. Eldawy and M. Mokbel (2013). A demonstration of Spatialhadoop: an efficient mapreduce framework for spatial data. In Proc. VLDB, 6(2), 1230-1233, 2013.
[3] Y. Hu et al (2012). Topological relationship query processing for complex regions in Oracle Spatial. In Proc. ACM-GIS, 2012.
[4] S. Nobari et al (2013). TOUCH: in-memory spatial join by hierarchical data-oriented partitioning," in Proc. ACM SIGMOD.
[5] Cloudera, Impala, http://impala.io/.
[6] J. Zhang, S. You and L. Gruenwald (2014). Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs," Information Systems, vol. 4, p. 134–154, 2014.
[7] J. Zhang and S. You (2013). GPU-based Spatial Indexing and Query Processing Using R-Trees," in Proc. ACM BigSpatial workshop.
[8] Thrust Parallel Library, https://thrust.github.io/
[9] ESRI, http://esri.github.io/gis-tools-for-hadoop/.
[10] Apache, Spark, https://spark.apache.org/
[11] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[12] J. L. Hennessy and D. A. Patterson (2011). Computer Architecture: A Quantitative Approach, 5th edition. Morgan Kaufmann.
[13] J. Parri, et al (2011). Returning Control to the Programmer: SIMD Intrinsics for Virtual Machines," ACM Queue, 9(2), 30-37.
[14] J. Zhang and S. You (2014). Large-Scale Geospatial Processing on Multi-Core and Many-Core Processors: Evaluations on CPUs, GPUs and MICs. CoRR, vol. abs/1403.0802.
[15] LLVM, The LLVM Compiler Infrastructure, http://llvm.org/.
[16] WKT, http://en.wikipedia.org/wiki/Well-known_text
[17] B. Kirk and W.-m. W. Hwu (2010). Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann.
[18] NYC Taxi Trips, http://www.andresmh.com/nyctaxitrips/
[19] NYC Census Block dataset, http://www.nyc.gov/html/dcp/pdf/bytes/nycb2000wi_metadata.pdf.
[20] GBIF Data Portal, http://data.gbif.org
[21] WWF Ecoregions, http://www.worldwildlife.org/biomes
[22] Libspatialindex, http://libspatialindex.github.io/
[23] GDAL, http://www.gdal.org/
[24] GEOS, http://trac.osgeo.org/geos