High-Performance Spatial Join Processing on GPGPUs with Applications to Large-Scale Taxi Trip Data

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

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

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

ABSTRACT

Spatially joining GPS recorded locations with infrastructure data, such as points of interests, road network, land cover and different types of zones, and assigning a point with its nearest polyline or polygon is a prerequisite for trip related analysis, which is becoming increasingly important in ubiquitous computing. However, existing spatial databases and GIS are incapable of handling large-scale data. The poor performance of these systems that takes tens of hours to dozens of days to complete such a commonly used spatial join query is undesirable. By leveraging several high-performance techniques, including massive data parallel General Purpose computing on Graphics Processing Units (GPGPU) technologies and cache friendly main-memory data structures, we are able to design an efficient spatial join query processing system. The experiments using a commodity workstation equipped with a Nvidia GPU device and real NYC taxi trip location data show that our system can join 170 million points with nearly a million polygons based on the nearest neighbor principle in about 33 seconds The performance represents a 3-4 orders of speedup when compared with an optimized serial CPU implementation using two leading open source packages for spatial indexing and spatial data management, respectively. We report our designs and implementations of GPU based filtering and refinement in spatial join processing and discuss the implications of modern hardware architectures for spatial databases and GIS.

1. INTRODUCTION

Spatial joins are fundamental in Spatial Databases (SDB) and Geographical Information System (GIS). Given two geospatial datasets (which can be points, polylines and polygons), a spatial join finds all pairs of objects satisfying a given spatial relationship between the objects, such as within, intersect and nearest neighbor. Spatial joins on CPUs have been extensively studied over the past few decades [1] given their practical importance. However, while research in parallel spatial joins can be dated back to 1990s [2-4], it was not until General Purpose computing on Graphics Processing Units (GPGPU) technologies on commodity hardware become available in recent years that using parallel spatial join processing to speed up SDB and GIS performance starts to be practical, both technologically and economically. This is because in the past accesses to parallel computing resources, such as supercomputers and grid computing facilities, were very limited to general users in the geospatial computing community [5]. The current commodity GPU architectures closely resemble supercomputers as both implement the primary Parallel Random Access Machine (PRAM[1]) characteristic of utilizing a very large number of threads with uniform memory latency [6]. The powerful parallel hardware architectures and the availability of low cost commodity GPU devices that are capable of general computing make it attractive to use GPUs for geospatial computing [7], including exploring the new designs and implementations of spatial joins on the new hardware for practical applications which is the purpose of this study.

The work reported in this paper is motivated by a practical large-scale spatial data management problem in associating hundreds of millions of points (taxi pickup locations at a yearly scale) with hundreds of thousands of polygons (tax lots whose land use types serve as a proxy for trip purposes) based on the nearest neighbor principle in the New York City (NYC). As the research in identifying trip purposes from large-scale taxi trip data is exploratory in nature, a fast implementation that can provide near real-time responses is essential. While nearest neighbor based spatial joins is supported in several spatial databases (e.g., Oracle Spatial) and GIS (e.g., ESRI ArcGIS), the response times on a single processor are too long to be practically useful when processing data at such a scale. Our experiments have shown that DBMS overheads and inefficient disk I/Os could prolong the processing times to tens of days. Furthermore, mainstream spatial databases and GIS do not natively support parallel hardware architectures yet. An alternative solution is to explore parallel processing power in Cloud Computing by adopting a MapReduce[2] parallel computing framework and incorporating open source GIS packages. Although a few successful stories have been reported [8, 9], we have several concerns in adopting the approach for our application that require near-interactive responses.

First, similar to using disk-resident spatial databases and GIS, disk I/Os can be a severe bottleneck in achieving the desired response time as the Hadoop Distributed File System (HDFS[3]) and its alike require signficant reading and writing accesses to slow disks and network channels to realize the parallelization. It is not clear whether it is possible to achieve the level of scalability for interactive applications due to the disk I/O and network bottlenecks. Second, even though it might be economically viable to execute MapReduce jobs in a Cloud Computing environment, developing and debugging MapReduce systems in a distributed environment can be difficult, time consuming and economically expensive. Third, previous works have shown that MapReduce systems are inefficient in utilizing computing resources [10]. Given that our reference serial implementation using open source geospatial data management packages (more specifically libspatialindex[4]

for R-Tree based polygon indexing and GDAL/OGR[5] for distance computation between points and polygons) required more than 30 hours (see Section 4 for details), to achieve a response time at the 10-100 seconds level, a speedup of 3-4 orders (1,000X-10,000X) is needed. Even assuming that Cloud Computing can achieve a linear speedup, without fundamental changes on data structures, algorithms and system implementations, this would be a huge waste of energy even if it is economically affordable.

As such, we have decided to explore GPGPU computing technologies to gain the desired level of performance. By integrating system provided parallel primitives (more specifically the Thrust[6] library) and our in-house developed modules, we are able to develop a working system that allows spatial join of 170 million taxi pickup locations with nearly a million polygons in NYC area in 2009 in about 33 seconds. This represents a more than three orders (3,325X) speedup compared with the reference optimized serial implementation, both run on a single commodity workstation. Three multiplicative factors, including *collective query strategy, in-memory data structure* and *parallel hardware*, have contributed to the significant speedup. First, while traditionally spatial indexing is considered expensive and some spatial join algorithms have been designed for non-indexed data to achieve better performance [1], we have developed a fast quasi-indexing approach on GPGPUs to quickly assign points to quadrants and use the point quadrants as the basic units for the spatial join. Since many points are spatially close to each other in our point dataset and very often nearby points will be joined with a same polygon, it is beneficial to use the quadrants instead of the individual points to query against indexed polygons. We term this strategy as *Collective Query*. The saving can be signficant when the numbers of points in the quadrants are large. Second, instead of using sophisticated data structures that requires dynamic memory allocations, which become increasingly expensive on modern hardware [11], simple linear data structures (including arrays and vectors) are used to stream data efficiently from disks to CPU main memories and to GPU global memories. The design is cache friendly and reduces TLB (Translation Lookaside Buffer[7]) misses [12, 13]. Using Structures of Arrays (SoA) instead of Arrays of Structures (AoS) further increases cache hits as more relevant data can be loaded into a cache line for small (short-length) data items. Third, after point quadrants are paired with polygons, computing the distances among all points in a quadrant and the polygon segments become embarrassingly parallelizable and is more suitable for GPUs due to its floating point computing power and high bandwidth between off-chip global memory and on-chip registers. A similar framework has been adopted in our previous work on spatially joining points and polygons through point-in-polygon test [14]. In this study, we focus on investigating the relative contributions of each of the three performance boosting technologies. As detailed in Section 4, these three factors contribute about 3.7X, 37X and 24X, respectively, which brings the combined speedup to 3,325X.

Although the signficant speedup is certainly practically useful, we believe it is more important to understand the limiting factors in the existing spatial databases and GIS software in achieving their potentials on modern parallel hardware architectures and we expect this work can serve as a case study for this purpose. Given the increasingly powerful (but not much about being faster on individual processors) parallel hardware and the increasingly large scale data, we believe it is timely to re-examine the cost models in designing spatial data structures and query processing algorithms in spatial databases and GIS. In particular, while existing spatial databases are mostly designed to be disk-resident and transaction oriented, there are increasingly large-scale OLAP[8] type applications on read-only spatial data where GPU accelerations can be instrumental. Existing indexing structures are heavily tailored for serial implementations on CPUs which seems to favor sophisticated designs in minimizing computation. However, different from the hardware invented 20-30 years ago, computation now is almost free while memory access can be hundreds of times slower in terms of clock cycles on modern hardware [15]. Despite many Software Development Kits (SDKs) are designed to purposely hide hardware details to improve programming productivity, a hardware-software co-design approach is essential to achieve the desired high performance. A key issue in this initiative is to identify the inherent data parallelisms in geospatial data processing and map it to parallel hardware in an appropriate way. Our implementation has adopted a parallel-primitive-oriented approach by using system-provided generic parallel primitives as much as possible and developing new parallel primitives for multidimensional data. We hope this design and development effort can eventually lead to a parallel primitive library designed for geospatial data to help make traditional spatial databases and GIS adaptive to parallel computing environments.

Our technical contributions are the following. First, we have designed and implemented a system to spatially join large-scale point locations with polygons completely on GPUs based on the nearest neighbor principle within an r-expanded window (or Windowed NN join), including both the filtering phase (by adopting a grid file based indexing approach) and the refinement phase (by assigning point quadrant and polygon pairs to GPU computing blocks for pair wise distance computation). Second, by integrating several high-performance computing technologies, we have reduced the computing time from 30.5 hours to 33.1 seconds and achieved more than 3,200X speedup when joining the pickup locations of 170 million taxi trip records with nearly a million tax-lot polygons. The system has made interactive spatial queries possible for the data at this scale. Third, by hybridizing GPU and CPU implementations, we have investigated the relative contributions of the speedup of the three performance boosting techniques and discussed their implications in designing high-performance SDB and GIS.

The rest of the paper is arranged as follows. Section 2 introduces the background, motivation and related work. Section 3 describes system design and implementation details. Section 4 presents the experiments and evaluations. Finally Section 5 provides conclusion and future work.

2. BACKGROUND, MOTIVATION AND RELATED WORK

Computing has evolved into a parallel era. In fact, from high-end servers to smart phones, it becomes increasingly difficult to find uni-processor based devices. Unfortunately, the software industry in general and the data management system

vendors in particular are slow in adapting to the parallel era and making full use of parallel hardware for BigData[9] applications. Geospatial computing (including both data management and analytics) are inherently data intensive. Although geospatial data are mostly collected and distributed by government agencies in the past (such as satellite imagery and urban infrastructure data), ubiquitous location and sensing data generated by individuals using handheld devices, such as GPS traces [16], cell phone call logs [17], location dependent social networks data [18, 19] and location enhanced photos and videos [20], are becoming increasingly popular. The data volumes have been growing exponentially. These new types of geospatial data are essential in understanding natural and social dynamics at community, city, national and global scales, especially when they are associated with infrastructure data and domain knowledge. There are several unique features in such data. First, they are mostly point data (e.g., GPS readings) or can be represented as point data after proper transformations (e.g., geocoding). Second, these data are highly concentrated in urban areas where human activities are high. Third, infrastructure data, such as road networks, Points of Interests (POIs) and land use types are crucial to understand the underlying meaning of the data, such as trip purposes and social interactions.

Although GIS and SDB are the commonly used tools to handle geo-referenced spatial data, we argue that existing GIS and SDB technologies and available tools are inefficient and/or insufficient in managing large-scale ubiquitous urban sensing data. The object-relational data models that are utilized in most of existing GIS and SDBs, while generic to handle many types of geospatial data, are not efficient to process point data at levels of hundreds of millions. The complex software structures also make it difficult to adapt to modern hardware, including parallel processing units, large-memory capacities and evolving cache hierarchies. The tuple-oriented physical data layout also makes it inefficient in processing read-only data when compared with column-oriented data layout for fast in-memory processing [21]. Towards this end, we have designed a prototype system called U^{2}SOD-DB [22] that is targeted at managing large-scale ubiquitous urban sensing origin-destination data. We next briefly introduce the layered functional modules in U^{2}SOD-DB to put the work reported in this paper into the context.

At the bottom layer, U^{2}SOD-DB adopts a time-segmented, column-oriented physical data layout to support in-memory, array-based data structures and allow fast streaming among disks, CPU memories and GPU global memory. The middle layer provides essential functionality on data compression, indexing and spatial/temporal aggregations. The top layer of U^{2}SOD-DB is designed to support more application-oriented operations such as joining point locations with points (e.g., POIs), polylines (e.g., road segments) and polygons (e.g., land use lots and census blocks) and shortest path computation. U^{2}SOD-DB targets at supporting the three types of spatial joins between point locations and urban infrastructure data, namely P2N-D join to snap a point to its nearest street segment, P2P-T join to associate a point with a polygon that the points falls into and P2P-D join to associate a point with a polygon that the point is closest to the polygon. Obviously, operations in this layer are much more computationally intensive and will benefit from performance boosting techniques including cache friendly in-memory data structures and parallelization on GPGPU accelerators. The work reported in this study is a new implementation of the P2P-D module of U^{2}SOD-DB where spatial join on points and polygons is based on the nearest neighbor principle.

Our previous results have shown that, for P2P-D spatial join, we were able to reduce the runtime from 30.5 hours to 1000-1500 seconds by adopting a hybrid approach where the points are quasi-indexed on GPUs and spatial join is performed on CPUs. An impressive 100X+ speedup has been achieved [22]. In this study, we show that the GPGPU parallel accelerations have brought a 26.5X speedup for distance computation which is the dominating component in the spatial join. This in turn brings the overall speedup to 3,325X when joining 170 million points with 735,488 real-world polygons. Due to the modular design, we are able to replace GPU parallel modules with functionally equivalent serial CPU modules wherever necessary to investigate the relative contributions of the three performance boosting techniques that are mentioned earlier. The experiment results have provided us some solid evidence in suggesting viable paths in making SDB and GIS software adaptive to the new parallel era. The proposed spatial join implementation reuses several components of U^{2}SOD-DB, including physical data layout, parallel indexing of point data and an ETL[10] (Extract, Transform and Load) module to convert polygon data to arrays. It is our goal to synergize these inter-related components which eventually leads to a high-performance geospatial data management system on commodity parallel devices without extra costs. In addition to core algorithm design and implementations, there are considerable software engineering issues need to be solved before the system can be reliably used for practical applications.

There is a rich body of related works on parallel algorithms and their GPGPU implementations, spatial indexing and spatial joins, and, urban computing applications of geospatial analysis While it is beyond our scope to provide a comprehensive review on the related works in the respective research areas, here we discuss a few related works that are most relevant to our research and development efforts. Research on spatial indexing and spatial joins using parallel primitives can be dated back to the seminal work by Hoel and Samet [2]. They have designed and implemented spatial operations using parallel primitives for PMR Quadtree trees and R-Trees and used them for operations such as polygonization based on topologies and spatial joins based on spatial intersections. Vertices and line segments were used as the basic units for spatial indexing and operations. In addition to targeting at the different hardware architectures (then supercomputer CM-5 v.s. current commodity shared-memory GPUs), our parallelization focuses on distance computation and nearest neighbor assignment which requires a different sets of parallel primitives from a conceptual design perspective. Nevertheless, our decision on using a simple grid-file structure to index Minimum Bounding Boxes (MBRs) of both point quadrants and polygons is partially motivated by their results showing that regular disjoint decompositions (PMR Quadtree) performed better than irregular decompositions (R-Trees) in a parallel setting. As detailed in the next section, when both MBRs of point quadrants and polygons are mapped to a simple grid, pairing two MBRs from the two joining datasets can be transformed into an equijoin problem on grid cell identifiers which can be efficiently

supported by combining generic parallel primitives such as *sort*, *reduce, binary_search* and *unique*. Another difference is that while we try to make full use of parallel primitives that are available in the underlying software development system (CUDA SDK[11] in this case), we realize that operations on multidimensional data can not always be efficiently (or even possibly) implemented on top of generic parallel primitives that are designed for 1D vectors. As such, we have implemented a set of functions that are required by our application (but are not supported by the underlying parallel library) using native parallel programming languages (CUDA C in this case).

Another category of related work is relational joins on GPUs. The GDB prototype system [23] has provided implementations of different types of relational joins, such as nested loop joins (both indexed and non-index), sort-merge joins and hash joins. A more complete set of relational algebra algorithms have been implemented recently [24] and reportedly better performance has been achieved. The work reported in [25] is interesting in the sense that, instead of transferring all relevant data to GPUs before relational joins, some data are designed to reside on CPU main memory and are exchanged dynamically between the CPU and GPU memories by the GPU-based join program through hardware supported DMA[12]. The relational join implementation matches GPU hardware architecture very well and the performance is impressive. However, spatial joins are significantly different from relational joins as pairing spatial objects in the two joining datasets are based on multidimensional spatial relationships instead of simple equality test (for natural joins) or evaluating a boolean expression (for theta joins). Furthermore, unlike relation joins that all tuples are flat and pairing two tuples can only be evaluated to true or false, pairing two spatial objects in different hierarchies may generate an unknown number of pairs which makes it rather difficult to parallelize.

While there are several pioneering works in spatial indexing on GPUs in computer graphics research (e.g., [26]), we argue that they are mostly designed for computer graphics applications such as ray-tracing and iso-surface constructions. They may not be suitable for spatially joining multiple datasets that is commonly required in SDB/GIS. There are several attempts to implement the classic R-Tree spatial indexing structure on GPUs and the work reported in [27] seems to be the most comprehensive one. The authors have tested parallel spatial range queries on constructed R-trees on GPUs which can be potentially modified for spatial join by treating the independent geometric objects used for queries as the non-indexed source dataset to be joined. However, while R-Trees have been extensively used on CPUs for spatial join, it is not clear whether R-Trees are good choices for spatial joins on GPUs. This is because data accesses are highly irregular when pairing the bounding boxes of geometrical objects that are indexed by R-Trees in both the source and target datasets that participate a spatial join. The problem has also been observed in previous research on all-pair nearest neighbor queries on CPUs [28].

One of our technical contributions in this paper is to tessellate spatial objects into sets of basic units according a certain spatial ordering and then use them as the intermediate tuples to apply relational joins on GPUs in the filtering phase of spatial join, i.e., use the grid cells as the basic units for equal joins to emulate spatial joins. The design utilizes a simple grid file data structure which might be superficially similar to the in-memory grid file data structure on GPUs first proposed in [29] as both are derived from classic grid file structures. However, there are several key differences between the two. First of all, the grid files proposed in [29] are designed to process exact match or range queries while our grid file is designed to process spatial join. Second, the grid file in [29] is used to index points directly while our grid file is used to index bounding boxes of both point quadrant and polygons (detailed in Section 3). It would be impossible to index hundreds of millions point directly on GPUs due to the memory capacity constraints. Third, while their range queries locate points within query windows directly without needing further processing, our spatial join computes unique pairs of point quadrants and polygons which requires complex post-processing including sorting, searching and removing duplicates.

3. SYSTEM DESIGN AND IMPLEMENTATION DETAILS

3.1 Overview

Following the general procedure of spatial joins [1], our data parallel spatial join algorithm also has two phases: the filter phase and the refinement phase. In the filter phase, we identify the largest quadrants that have less than K points (quasi-indexing) and build a grid-file based index structure for the bounding boxes or expanded bounding boxes of both the point quadrants and the polygons to be joined. The algorithm identifies (qid, pid) pairs that potentially satisfy the spatial join criteria where qid represent an identified quadrant and pid represents a polygon. In the refinement phase, distances among all the points in the quadrant and all the edges of the polygon are computed to assign all the points with the polygon ids of their nearest polygons. Here the distance between a point and a polygon is canonically defined as the minimum distance among all the distances between the point and edges (or segments) of the polygon. The overall framework is similar to what we have designed for point-in-polygon test based spatial join [14] with one major difference. Instead of using MBRs of polygons directly, we expand the polygon MBRs with r units along the width dimension and r units along the height dimension (the purpose of the modification is detailed in Section 3.3). The modified framework for point-to-polygon spatial join based on the nearest neighbor principle (or P2P-D for short) is illustrated in Fig. 1. In the following subsections we present the details of the components, i.e., in-memory data structure, quadrant identification, associating bounding Boxes with grid cells, and, pair-wise distance computation and polygon assignment.

3.2 In-Memory Data Structures and Quadrant Identification

The details of the in-memory data structures are discussed in [14][22] and the details of generating point quadrants are provided in [14]. For the purpose of being self-contained, we next sketch the key points of these two techniques that are the integral components of the system. As shown in the top-right part of Fig. 1, instead of using variable-sized data structures to store polygon data which requires dynamic memory allocation in many existing SDB and GIS, we flatten the complex data structures into five one-dimensional arrays;

the *Feature Index* array which stores the positions of the first polygon features in polygon datasets, the *Ring Index* array which stores the positions of the first rings of all polygon features and the *Vertex Index* array which stores the positions of the first vertices for all rings, and finally, the *X/Y Coordinate* arrays which store the x/y coordinates for all rings. The design closely reflects the OGC SFS specification[13] which has been adopted by most commercial and open source GIS/SDB software. As a main-memory based high-performance system, we assume all the arrays are memory-resident and can be streamed among hard drives, CPU memories and GPU memories as chunks. For the 735,488 polygons used in our experiments, three are 4,698,986 vertices which take just a few tens of megabytes and can be easily accommodated by commodity computers.

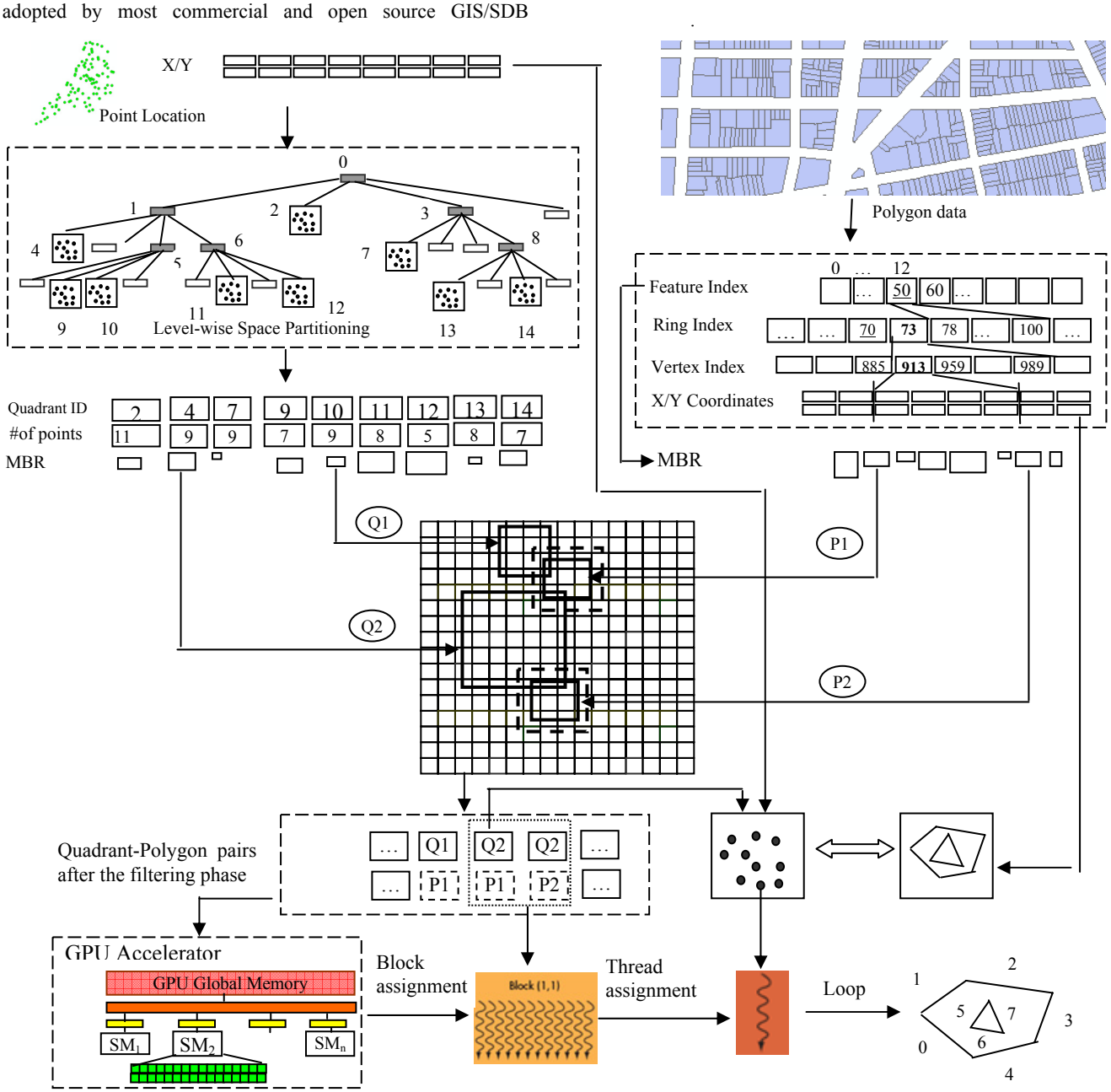

Fig. 1 Framework of System Design

The algorithm to generate point quadrants with no more than K points on GPUs originates from our work on building Constrained Spatial Partition Trees for Point Data (CSPT-P) [30] for load balancing purposes in processing geospatial data, such as GPU-based Geographically Weighted Regression (GWR) [7]. In this study, we follow a similar approach that has been exploited in our previous work on point-in-polygon test based spatial join [14], i.e., re-using the first part of the CSPT-P construction algorithm for identifying leaf tree nodes to generate point quadrants. This is based on the observation that only leaf nodes and not the complete space partitioning tree is needed for spatial join using a flat grid file for filtering and we term it as quasi-indexing. That being said, the whole CSPT-P construction algorithm can be modified to generate multi-level grid file data structures or used directly for the filtering phase of the spatial join which is left for our future work.

Basically a CSPT-P tree spatially divides a point dataset into quadrants and marks quadrants with no more than K points as leaf nodes. When a CSPT-P tree is constructed hierarchically in a top-down manner, there are considerable data parallelisms at each level and thus parallel primitives, including *copy*, *transform*, *sort*, *scan*, *gather*/*scatter* and *reduce*, can be used efficiently and effectively to implement the tree construction algorithm on GPUs. First, points are sorted based the Morton codes[14] which are derived from their coordinates at a level (*transform* + *sort*). Second, the numbers of points that fall within each quadrant can be computed (*reduce* by key). Third, points that fall within the quadrants that have fewer than K points can be shifted to the beginning of the input point vector and the boundaries of the identified quadrants can be marked (using a combination of *copy*, *scatter*, *scan* and *gather*). Finally, the bounding boxes of the quadrants are computed (*transform* + *reduce* by key). While it is beyond the scope our study to present the details of primitives based parallel programming, we refer to the appendix (located at the last page) of [14], which provides a brief introduction to the parallel primitives that have been used in this study. The examples given in the appendix may help understand the functionality of the respective parallel primitives.

3.3 Associating Bounding Boxes with Grid Cells

As shown in Fig.2, there are two possible ways to pair a point quadrant whose MBR is (x1,y1,x2,y2) with a polygon where points in the quadrant can potentially be paired with the polygon, i.e., the minimum distance between a point and the polygon is less than r. The first approach is to expand the MBR of the point quadrant. If the expanded bounding box of the point quadrant, i.e., (x1-r,y1-r, x1+r, y1+r), intersects with the MBR of the polygon then the point quadrant and the polygon is paired for further refinement. The second approach is opposite to the first approach by expanding the MBR of the polygon and finding the point quadrants that intersect with the expanded MBR of the polygon. We have chosen the second approach and the reasons will be provided shortly after we introduce the high-level design in pairing point quadrants and polygons.

To pair point quadrants with polygons, we rasterize the MBRs of point quadrants and the expanded MBRs of polygons to a uniformed grid and use equijoin on the cells to emulate the spatial join of the MBRs. It is clear that, after the rasterization, some of the grid cells will be covered by one point quadrant MBRs while some of the grid cells will not be associated with any point quadrant MBRs. The relationship between quadrant MBRs and grid cells is 1:n. On the other hand, since the expanded MBRs of polygons may overlap, the relationship between polygon MBRs and grid cells is m:n. Assume there are *s* point quadrant MBRs and *t* expanded polygon MBRs that are mapped to a same grid, then there will be s*t (qid, pid) pairs to be refined in the refinement phase. Note that neighboring grid cells might have similar (qid, pid) pairs and there are duplicated (qid,pid) pairs across grid cells (to be detailed next). Since only one unique copy of the pairs is needed to be refined in the refinement phase, the duplications should be removed before the refinement phase. This can be efficiently implemented using the *sort+unique* parallel primitives that are supported in the Thrust parallel library (and perhaps other parallel libraries). Our approach essentially transforms a spatial intersection test problem into a number of equality test problems, which are very basic and are well supported by all parallel hardware instruction sets.

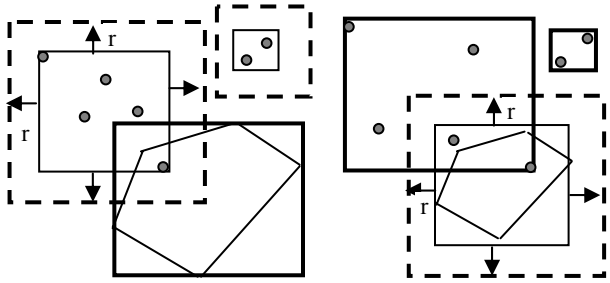

Fig. 2 Illustration of the Two Approaches in Pairing Points and Polygons through MBR Expansion

While conceptually the two approaches in pairing point quadrants with polygons discussed earlier are the same, we have found that the second approach has a much better performance when spatially joining taxi pickup locations with tax lot polygons due to the grid file based filtering approach and the distributions of the datasets to be joined. The main reason is that the majority of taxi pickup locations are clustered near the street intersections and the identified quadrants often have a large number of very small quadrants (e.g., from 2*2 feet to 32*32 feet). If r (e.g., 100 feet) is much larger than the quadrant sizes, the MBRs of these quadrants have large expansion ratios. When each expanded MBR is used to pair with a polygon MBR, neighboring point quadrants are likely to generate a large number pairs of quadrant identifiers and polygon identifiers, or (qid, pid) pairs, with considerable duplications. This has been illustrated in Fig. 3. While there are only a few pairs that should be identified, e.g., {(Q2, P2), (Q2, P3), (Q3, P1), (Q3, P2), (Q3, P3), (Q4, P1), (Q4, P2), (Q4, P3)}, each of the expanded MBRs of the four point quadrants are rasterized into 9*9 cells due to the large r value. In contrast, if we expand the MBRs of the polygons (not shown in Fig. 3), we will still get the same pairs but far fewer cell identifiers would be involved. In addition to requiring large temporary GPU memories to hold the intermediate cell identifiers for both the rasterized MBRs and

for matched (qid, pid) pairs before removing duplicates, excessive memory accesses to write and read (qid, pid) pairs can also reduce overall system performance in our parallel primitives base implementation. As such, we have decided to adopt the second approach and expand the MBRs of polygons.

What needs to be explained next in more details is the exact procedure of mapping a box (for both a point quadrant MBR and an expanded MBR of a polygon) to a grid. Given a vector of quadruples (x1, y1, x2, y2) for the input MBRs and a grid cell size cz, we wish to output the cell ids in the range of col=floor(x1/cz)..ceiling(x2/cz) along the x dimension and row=floor(y1/cz).. ceiling(y2/cz), where col and row can be combined using any Space Filling Curve (SFC) ordering[15] and here we use row-major order for simplicity. While this seems to be trivial for a serial implementation, it turns out to be nontrivial for a parallel computing model. The biggest issue is to let the parallel processing units (i.e., threads) know the exact locations to get the input data from and the exact locations to write the output data to. A similar effort to compute the locations using parallel primitives took more than 20 steps (see page 6 of our technical report at [31] for an example) which has limited the GPU speedup to only 6.7X for the refinement phase of distance based spatial join. As such, in this study, we have decided to develop a parallel primitive directly on top of CUDA for this purpose. The procedure first uses a prefix-sum over the numbers of cells for all MBRs to be mapped so that each thread knows where to start outputting the computed cells. Second, within the CUDA kernel to compute the cell ids, a thread loops over rows and columns of a rasterized box and writes out both the quadrant identifier (qid) and the cell identifiers (cids) to which it is mapped. While the kernel is not very efficient in the sense that memory accesses are not coalesced and the workload is not balanced among threads either, we have found that the performance is better than that of the implementation using generic parallel primitives by avoiding computing and outputting a large number of intermediate positions that are needed for thread coordination [31].

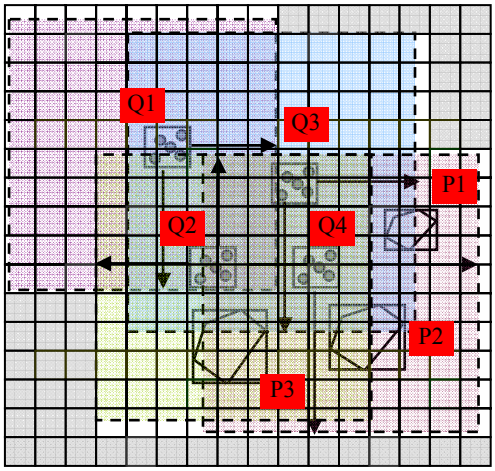

Fig. 3 Illustration of the Problem in Expanding MBRs with a Large Expansion Distance

We next provide implementation details on GPU based association between quadrant identifiers (qids) and polygon identifiers (pids) through the grid cells (cids) to form (qid, pid) pairs for refinement. The high-level conceptual design has been discussed earlier in this subsection. Recall that the cell mapping kernel generates vectors for MBRs of both the point quadrants and expanded MBRs of polygons with the first vector storing qids or pids and the second vector storing the corresponding cell identifiers (cids). Let us call the two vectors for the point quadrants are VQQ and VQC, respectively. Similarly, two vectors are used for polygons, i.e., VPP, and VPC, as shown in the top-right part of Fig. 4. Our algorithm searches each of VPC elements in VQC and pairs the corresponding elements in VPP with the corresponding elements in VQQ if a search on the VPC element is successful. This can be efficiently implemented using a combination of the *binary_search* and *lower_bound* parallel primitives provided by the Thrust library (the details of the combination is available in the appendix page of [14], if needed). We note that searching VPC elements can be done in parallel by using the vertorized versions of the two primitives.

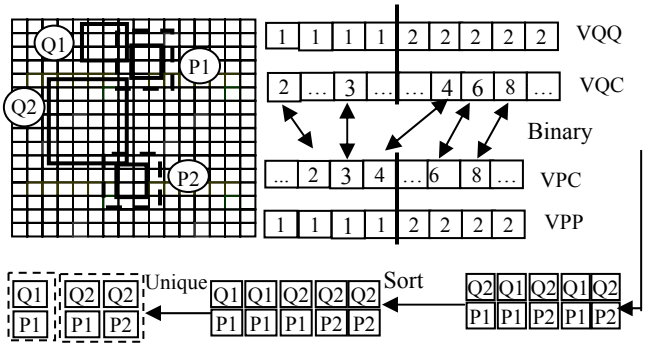

Fig. 4 Illustration of Grid-File based Spatial Join Filtering and its GPU implementation using Parallel Primitives

3.4 Pair-wise Distance Computation and Polygon Assignment

After the filter phase is completed, a subsequent refinement phase is followed. In this phase, pair-wise distance computation among all the points in the quadrant (identified by the qid) and all the segments in a polygon (identified by the pid) are computed to assign points with the proper polygon identifiers based on the nearest neighbor principle. From a parallel computing perspective, the parallelism in this step is very similar to rasterizing MBRs as we have discussed in Section 3.3. For the same reason (excessive overheads in computing and outputting locations in a parallel primitives based implementation), we have developed a CUDA kernel for this purpose.

It is expected that this step is very computationally expensive and can be the bottleneck of the overall performance. It is thus desirable to make full use of the parallel processing power of GPGPUs. We map the following two levels of data parallelisms to the CUDA computing model for this purpose. We assign a (qid, pid) pair to a computing block and assign a point (within a quadrant) to a thread (within a computing block) due to the following considerations. First, while each quadrant has fewer than K points to guarantee load balance to a certain degree, our empirical results have shown that the actual numbers of points in quadrants can vary significantly which may still incur load unbalance with a factor anywhere between 1

and K. Second, while the majority of the polygons in our applications have fewer than 20 vertices, a small portion can have large numbers of vertices. The skewness of real datasets makes it difficult for perfect load balancing. Fortunately, CUDA adopts a task-based scheduling approach at the computing block level. When there are sufficient computing blocks to be executed, the GPU hardware can be fully utilized even if the workloads among computing blocks are not well balanced. As such, it is appropriate to expose the bounded skewness at the computing block level. On the other hand, CUDA requires perfect load balancing among the threads within a computing block due to its SIMD (Same Instruction, Multiple Data) design. Since CUDA compute capacity 2.0 and above has a warp size of 32 which is larger than the numbers of segments of most polygons in our dataset, the hardware can not be fully utilized if we assign polygon segments to threads and let each thread loop through all the points. As such, it is natural to assign threads to points and let each thread loop through all the polygon segments. In fact, we can purposely set K to be divisible by the number of threads in a computing block (N) and use N in the distance computing kernel to make a good match. Since memory access patterns play a predominantly important role in achieving good performance in data-intensive GPU computing, we next discuss how data are accessed and how memory coalescing is achieved.

When a thread loops through rings and vertices of a polygon (c.f. Section 3.1), it first retrieves the starting and ending ring indices from the *Ring Index* array. For each of the ring index, it retrieves the starting and ending vertex indices from the *Vertex Index* array. As two consecutive vertices define a polygon segment, the distance between the point assigned to the thread and the segment can be computed. Note that when the point is perpendicularly projected to the extension (instead of falling onto) of the segment, the smaller distance to the two vertices of the segment is used instead. As points within a quadrant are paired with a same polygon, threads within a computing block will access the same set of vertices in all steps of the distance computation and thus memory accesses are perfectly coalesced.

While conceptually it is possible to collaboratively load the vertices to GPU shared memory by all the threads in a computing block to avoid multiple accesses to the vertices in GPU global memory, our experiments have shown that the performance gain is not signficant. There are three possible reasons. First, the memory accesses are already coalesced as we just discussed. Second, the unified L2 cache introduced in the Fermi architecture[16] may have cached the polygon vertices. Third, the distance computation kernel is computing intensive and the memory accesses overheads are overshadowed. By avoiding using shared memory, the implementation becomes more flexible in adjusting numbers of threads in a computing block (e.g., based on K values). In addition, the saved fast on-chip shared memory can potentially be used to store per-thread information which is currently represented by registers. Due to the complexity of the distance computation kernel, our current implementation requires more registers than the GPU hardware can provide when GPU Stream Multiprocessors (SMs) are fully utilized (32768/1024=32 registers per thread) and the hardware occupancy is not full. By balancing the utilization of registers and shared memory, we expect to improve hardware occupancy and further improve the performance of the implementation. This is left for our future work.

4. EXPERIMENTS AND RESULTS

4.1 Data and Experiment Setup

Through a partnership with the New York City (NYC) Taxi and Limousine Commission (TLC), we have access to roughly 300 million GPS-based trip records for a duration of about two years (2008-2010). Each taxi trip has a GPS recorded pickup location and a drop-off location expressed as a pair of latitude and longitude. In this study, we use the approximately 170 million pickup locations in 2009 for experiments. The polygon data we use is the NYC MapPluto tax lot data[17]. There are 735,488 tax lot polygons in NYC with 4,698,986 vertices. All experiments are performed on a Dell Precision T5400 workstation equipped with dual quadcore CPUs running at 2.26 GHZ. The workstation has 16 GB memory and a 500G hard drive and is equipped with a Nvidia Quadra 6000 GPU device with 448 CUDA cores and 6 GB GDDR5 memory. The sustainable disk I/O speed is about 100 megabytes per second while the theoretical data transfer speed between the CPU and the GPU is 4 gigabytes per second through a PCI-E card.

For comparison purposes, we have implemented the same spatial join using open source GIS packages, i.e., libspatialindex to index polygon data by building an R-Tree, and, GDAL, which implicitly uses GEOS, to perform the point-to-polygon distance computation. While we could have also built an R-Tree for the point locations by treating the points as bounding boxes, given the large number of points, it is very costly to index the point data using R-Tree indexing. In addition, coordinating the two index structures to perform the spatial join is non-trivial on CPUs and is beyond the scope of this research. As such, the CPU implementation computes the distance from each point to the polygons whose MBRs intersect with the point based on the R-Tree indexing and chooses the one that has the smallest distance. We compiled both the CPU and GPU source code with –O2 optimization flag for fair comparison.

Our GPU implementation has two major parameters to set. The first parameter is K, the maximum number of points in a quadrant and the second parameter is the number of threads per computing block (N) in the pair-wise distance computing kernel. We have tested three K values (256, 512 and 1024) and three N values (128, 256 and 512) under the condition that K>=N. We first report the best GPU result and compare it with the CPU implementation. We then provide details on the GPU implementation.

4.2 Overall Results

The overall results are summarized in Table 1 where the four implementations, i.e., All CPU–baseline, Hybrid-1, Hybrid-2 and All-GPU, adopt the three performance boosting techniques as marked. The parameters have been set to K=256, N=256 based on the experiments and r is empirically fixed to 100 feet (~30 meters) for the GPU implementation where the best results are achieved. The CPU implementation also uses K=512 and r=100 feet (N is irrelevant) for comparison purposes. From Table 1 we can see that the three performance boosting techniques have contributed 3.66X, 37.37X and 24.28X

speedups, respectively. This brings a total speedup of 3,325.1X. Among the three techniques, using the in-memory data structures seems to be the most signficant (37.37X). Since indexing points can also be done in external memory using traditional techniques and using in-memory data structures is the prerequisite for GPU accelerations, we also compute the combined speedup of the latter two techniques which are more related to modern hardware. The combined speedup is 907.5X and is close to 3 orders which may indicate that there are considerable potential in improving the performance of existing spatial databases and GIS software by using GPU accelerations and data structures that are friendly to modern hardware architectures.

4.3 Results of GPU Implementation

The runtimes of the GPU implementation include the following components: generating point quadrants (12.238 seconds), pairing point quadrants and polygons in the filtering phase (1.957 seconds) and pair-wise distance computation and comparison in the refinement phase (18.915 seconds). Clearly the refinement phase dominates the overall process. While there are quite some performance improvement opportunities for the filtering phase, we are more interested in optimizing the refinement phase. Our experiments have shown that using different combinations of K and N can nearly double the runtime of the refinement phase (with K=1024 and N=128) which warrants further investigation. Although using a large K will decrease the runtime of generating point quadrants, it will allow more points in a quadrant and thus the MBRs of such quadrants will be larger. Due to the reduced filtering power, there will be more distance computations needed to be done in the refinement phase. As such, the current implementation favors small K values with respect to runtime. However, when K is small (and hence large number of quadrants), there will be a large number of (qid, pid) pairs which may impose signficant memory pressure on GPUs. Furthermore, K can not be less than the GPU warp size set by CUDA compute capabilities (currently 32) in order to fully utilize GPU. As discussed in Section 3.4, since our GPU implementation uses more than 32 register files per thread, using large N will incur low occupancy rate which typically will bring down the kernel performance. On the other hand, using small N will need to perform multiple loops over K (note that we have set K to be divisible by N) and can incur synchronization overhead. As reported earlier, we have found that K=256 and N=256 achieved the best performance among all the combinations we have tested.

Despite the significant speedup over the serial CPU implementation using the state-of-the-art open source software packages, there are still considerable rooms for performance improvements and the limited GPU memory capacity issue (when compared with CPU) also needs to be addressed from a system development perspective. For example, using a multi-level grid file or tree indexing structures may reduce the memory requirement in the filtering phase by generating fewer (qid, pid) pairs which have been observed and addressed in our previous work [32, 33] from a spatial range query perspective in a CPU setting. Another improvement that can potentially significantly improve the performance of the refinement phase (which in turn will boost the overall performance due to Amdahl's law) is on-the-fly indexing of points and polygon segments within a computing block using GPU fast shared memory (and/or global memory). The work by Hoel and Samet [2] can be adapted for the GPU hardware architecture for this purpose as well although their technique was targeted for a very different parallel hardware architecture (the CM-5 machine). This is especially useful for large point quadrants and polygons where indexing is effective in pruning search space for spatial join. Subsequently a meta-module is needed to decide when to and when not to perform such on-the-fly indexing. These are left for our future work.

Table 1 Overall Results on Applying the Three Performance Boosting Techniques

Implementation/Performance boosting techniques applied	Collective query (for points)	in-memory data structures	GPU acceleration	Runtime (seconds)	Step-wise speedup (X)	Accumulative speedup (X)
All CPU (baseline)				11,0093.680	/	/
Hybrid-1	X			30,048.223	3.66	3.66
Bybird-2	X	X		804.016	37.37	136.93
All GPU	X	X	X	33.110	24.28	3,325.1

5. CONCLUSION AND FUTURE WORK

In this study, we have reported our GPGPU-based designs and implementations on spatially joining large-scale point location data with polygon data which is an important operation in spatial databases and GIS. The high-performance system can help identify trip purposes when applied to processing large-scale ubiquitous urban sensing data such as GPS recorded pickup and drop-off locations of taxi trip records. Experiments have shown that, with a combination of in-memory data structures, collective query strategy and GPU hardware parallel accelerations, we have achieved 3,325 times of speedup when compared to a baseline serial CPU implementation on top of the state-of-the-art open source GIS packages.

For future work, first of all, we would like to analyze the potential of further performance improvements. The majority of the current implementation is built on top of the Thrust parallel library which incurs some unavoidable duplicated computing. The kernels we have developed can also be optimized in terms of load balancing and algorithmic engineering (especially for the pair-wise distance computing kernel code) in addition to using more efficient indexing structures for filtering on GPUs and on-the-fly indexing of both points and polygon segments in a computing block. Second, we

plan to systematically address the limited GPU memory capacity issue by segmenting the inputs in the primitives-based modules into chunks and combining the chunked results into the final ones. This is in addition to developing new data structures that has smaller memory footprint. Finally, after cleaning up the source code and preparing some data generation code (the taxi trip data and some of the urban infrastructure data are not shareable at present), we plan to make our code open source. We believe a community effort is needed to make traditional GIS/SDB adaptive to a parallel era.

6. REFERENCES

[1] Jacox, E. H. and Samet, H. (2007). Spatial join techniques. ACM Transaction on Database System 32(1).

[2] Hoel, E. G. and Samet, H., 1994. Performance of Data-Parallel Spatial Operations. Proceedings of VLDB Conference.

[3] Brinkhoff, T., Kriegel, H.-P. and Seeger, B. (1996). Parallel Processing of Spatial Joins Using R-trees. Proceedings of IEEE ICDE Conference.

[4] Zhou, X., Abel, D. J. and Truffet, D. (1998). Data Partitioning for Parallel Spatial Join Processing. GeoInformatica 2(2): 175-204.

[5] Clematis, A., Mineter, M. et al., 2003. High performance computing with geographical data. Parallel Computing 29(10): 1275-1279.

[6] Hong, S., Kim, S. K., et al., 2011. Accelerating CUDA graph algorithms at maximum warp. Proceedings of the 16th ACM symposium on Principles and practice of parallel programming.

[7] Zhang, J. 2010. Towards Personal High-Performance Geospatial Computing (HPC-G): Perspectives and a Case Study. Proceedings of ACM HPDGIS workshop.

[8] Zhang, S., Han, J., Liu, Z., Wang, K. and Xu, Z. (2009). SJMR: Parallelizing spatial join with MapReduce on clusters. Proceedings of IEEE International Conference on Cluster Computing.

[9] Zhang, C., Li, F. and Jestes, J. (2012). Efficient parallel kNN joins for large data in MapReduce. Proceedings of EDBT Conference

[10] Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D. and Moon, B. (2012). Parallel data processing with MapReduce: a survey. SIGMOD Record, 40 (4), 11-20.

[11] Hennessy, J.L. and Patterson, D. A, 2011. Computer Architecture: A Quantitative Approach (5th ed.). Morgan Kaufmann

[12] John, C. and Kenneth, A. R. (2008). Data partitioning on chip multiprocessors. Proceedings of ACM DaMoN workshop.

[13] Cieslewicz, J. and Ross, K. A. (2008). Database Optimizations for Modern Hardware. Proceedings of the IEEE 96(5).

[14] Zhang, J. and You., S. (2012). Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. Technical report online at http://geoteci.engr.ccny.cuny.edu/pub/pipsp_tr.pdf

[15] Krste, A., Rastislav, B., James, D., Tony, K., Kurt, K., John, K., Nelson, M., David, P., Koushik, S., John, W., David, W. and Katherine, Y. (2009). A view of the parallel computing landscape, CACM. 52, 56-67.

[16] Zheng, Y., Liu, Y., Yuan, J. and Xie, X. (2011). Urban computing with taxicabs. Proceedings of ACM UbiComp.

[17] Calabrese, F., Colonna, M. et al. 2010. Real-Time Urban Monitoring Using Cell Phones: A Case Study in Rome. IEEE Transactions on Intelligent Transportation Systems 12(1): 141-151.

[18] Vasconcelos, M. A., Ricci, S., et al. 2012. Tips, Dones and Todos: Uncovering User Profiles in Foursquare. Proceedings of ACM WSDM Conference.

[19] Calabrese, F., Kloeckl, K., et al. 2008. WikiCity: Real-time Location-sensitive Tools for the City. In Handbook of Research on Urban Informatics: The Practice and Promise of the Real-Time City (Foth, M. eds) 390-413. IGI Global.

[20] Friedland, G., Choi, J., et al. 2011. Video2GPS: A Demo of Multimodal Location Estimation on Flickr Videos. Proceedings of ACM Multimedia Conference.

[21] Abadi, D. J., Madden, S. R. and Hachem, N. (2008). Column-stores vs. row-stores: how different are they really? Proceedings of ACM SIGMOD Conference.

[22] Zhang, J., Gong, H., Kamga, C. and Gruenwald L. (2012). U2SOD-DB: A Database System to Manage Large-Scale Ubiquitous Urban Sensing Origin-Destination Data. To appear in proceedings of ACM SIGKDD UrbComp workshop.

[23] He, B. S., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q. and Sander, P. V. (2009). Relational Query Coprocessing on Graphics Processors. ACM Transactions on Database Systems 34(4).

[24] Gregory Frederick Diamos, Wu, H., Lele, A. and Wang, J. (2012). Efficient Relational Algebra Algorithms and Data Structures for GPU. Technical report. Online at http://www.cercs.gatech.edu/tech-reports/tr2012/git-cercs-12-01.pdf

[25] Kaldewey, T., Lohman, G., Mueller, R. and Volk, P. (2012). GPU join processing revisited. Proceedings ACM DaMoN Workshop.

[26] Zhou, K., Hou, Q., et al. (2008). Real-Time KD-Tree Construction on Graphics Hardware. ACM Transaction on Graphics 27(5).

[27] Luo, L., Wong, M. D. F., et al. (2011). Parallel implementation of R-trees on the GPU. Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC).

[28] Chen, Y. and Patel, J. (2007). Efficient evaluation of all-nearest-neigbor queries. Proceedings of IEEE ICDE.

[29] Yang, K., He, B., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P. and Shi, J. (2007). In-memory grid files on graphics processors. Proceedings of ACM DaMoN Workshop

[30] Zhang, J. and Gruenwald, L. (2012). CSPT-P Tree Indexing on Large-Scale Point data using Parallel Primitives on GPGPUs. Technical report. Online at http://134.74.112.65/primcsptp/CSPTP_tr.pdf

[31] Zhang, J. (2012). Parallel Primitives based Spatial Join of Geospatial Data on GPGPUs. Technical report. Online at http://134.74.112.65/primspjion/PPSJ_tr.pdf

[32] Zhang, J., Gertz, M. and Gruenwald, L. (2009). Efficiently managing large-scale raster species distribution data in PostgreSQL. Proceedings of ACM GIS.

[33] Zhang, J. (2012) A high-performance web-based information system for publishing large-scale species range maps in support of biodiversity studies. Ecological Informatics 8: 68-77.

[1] en.wikipedia.org/wiki/Parallel_Random_Access_Machine

[2] http://en.wikipedia.org/wiki/MapReduce

[3] http://hadoop.apache.org/hdfs/

[4] http://libspatialindex.github.com/

[5] http://www.gdal.org/

[6] http://thrust.github.com/

[7] http://en.wikipedia.org/wiki/Translation_lookaside_buffer

[8] http://en.wikipedia.org/wiki/Online_analytical_processing

[9] http://en.wikipedia.org/wiki/Big_data

[10] http://en.wikipedia.org/wiki/Extract,_transform,_load

[11] http://developer.nvidia.com/cuda-downloads

[12] http://en.wikipedia.org/wiki/Direct_memory_access

[13] http://www.opengeospatial.org/standards/sfs

[14] http://en.wikipedia.org/wiki/Z-order_curve

[15] http://en.wikipedia.org/wiki/Space-filling_curve

[16] http://www.nvidia.com/object/fermi-architecture.html

[17] http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml