Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs

Jianting Zhang
Department of Computer Science
The City College of the City University of New York
New York, NY, 10031
jzhang@cs.ccny.cuny.edu

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

ABSTRACT

Point-in-Polygon (PIP) test is fundamental to spatial databases and GIS. Motivated by the slow response times in joining large-scale point locations with polygons using traditional spatial databases and GIS and the massively data parallel computing power of commodity GPU devices, we have designed and developed an end-to-end system completely on GPUs to associate points with the polygons that they fall within. The system includes an efficient module to generate point quadrants that have at most K points from large-scale unordered points, a simple grid-file based spatial filtering approach to associate point quadrants and polygons, and, a PIP test module to assign polygons to points in a GPU computing block using both the block and thread level parallelisms. Experiments on joining 170 million points with more than 40 thousand polygons have resulted in a runtime of 11.165 seconds on an Nvidia Quadro 6000 GPU device. Compared with a baseline serial CPU implementation using state-of-the-art open source GIS packages which requires 15.223 hours to complete, a speedup of 4,910X has been achieved. We further discuss several factors and parameters that may affect the system performance.

1. INTRODUCTION

Point-in-Polygon (PIP) test is an important computational geometry operation and has been widely used in Computer Graphics (CG), Spatial Databases (SDB) and Geographical Information Systems (GIS). As locating and navigation sensors (such as GPS, cellular, Wifi and their combinations) have been increasingly embedded in personal handheld devices, huge amounts of point locations have been generated. Very often these point locations need to be associated with different types of infrastructure data (such as administrative regions and census blocks) for various analysis purposes. This is typically done in a SDB or a GIS environment by joining the point dataset with the polygon dataset. The functionality has been well supported by major commercial and open source packages. However, traditional SDB and GIS are mostly designed to be disk-resident and run on a single processor. Despite sophisticated indexing approaches have been developed over the past decades to speed up the spatial join process (see [1] for a comprehensive review), joining hundreds of millions of points with tens of thousands of polygons can take dozens of hours which is far from desirable for interactive queries.

In this study, we aim at utilizing massively data parallel computing power provided by Graphics Processing Units (GPUs) using General Purpose computing on GPUs (GPGPU[1]) technologies to speed up large-scale PIP test based spatial joins. Following the general spatial join strategy in spatial databases [1], we have developed a simple grid-file [2, 3] based indexing approach on GPUs for both point data and polygon data in the filtering phase and implemented an efficient PIP test on GPUs in the refinement phase. Our experiments have shown that the end-to-end runtime in joining 170 million points with more than 40 thousand polygons is reduced from 54,819 seconds (more than 15 hours) using an open source implementation to 11.165 seconds and a signficant speedup of 4,910X has been achieved. The signficant speedup not only saves computing resources but also makes real time user interactions possible.

Our technical contributions are the following. First, we have developed an end-to-end, high-performance system to join large scale point locations with polygons on GPUs which can be applied to a variety of real-world data-intensive applications. Second, we have designed and implemented a set of algorithms that can efficiently index large-scale point data and pairing points and polygons in the filtering phase of the spatial join. Third, we have investigated the design choices and the impacts of key parameters for the PIP tests on GPUs in the refinement phase of the spatial join. Finally, we have demonstrated that the performance of traditional disk-resident spatial databases and GIS can be significantly improved by incorporating modern hardware features and GPU accelerations. The rest of the paper is arranged as follows. Section 2 introduces background and related work. Section 3 presents the GPU based spatial join framework and implementation details. Section 4 provides experiment results and discussions. Finally, Section 5 is the conclusions and future work.

2. BACKGROUND AND RELATED WORK

Geospatial data is pervasive in our everyday lives. While traditionally geo-referenced data are often collected, processed and distributed by government agencies (e.g., Census Bureau and Department of City Planning), as more and more personal handheld devices are equipped with locating and

[1] http://en.wikipedia.org/wiki/GPGPU

navigation capabilities by using Global Positioning System (GPS), cellular and Wifi technologies and their combinations, geo-referenced point location data becomes an important ubiquitous sensing data and the data volumes are increasing very fast. It is necessary to align these point data with infrastructure data to make sense out of the point locations. While spatial databases and GIS are the commonly used tools to process geo-reference data, they are not optimized to align large-scale point data with infrastructure data.

From a geospatial modeling perspective, aligning points to different types of geo-referenced infrastructure data can be abstracted as a spatial join problem. According to the Open Geospatial Consortium (OGC) Simple Feature Specification (SFS) [2], the SQL expression can be something like the following:

SELECT Point.ID, Polygon.ID WHERE ST_WITHIN (Point.geometry, Polygon.geometry)

When the polygons are spatially mutually exclusive (non-overlapping), a point can only be associated with a single polygon. The functionality is well supported by most spatial databases and GIS. Spatial indexing approaches can be applied to both point and polygon data to speed up query processing. While spatial join query processing usually works well for small data on a single CPU processor, we are not aware of existing systems that can take advantages of the multicore and many-core parallel hardware resources that are already available in commodity computers to speed up spatial queries (we refer to [4] for a review on geospatial computing on GPUs). Although relational data management on multicore parallel hardware architectures have been a hot research topic over the past few years (for reviews see [5, 6]), unfortunately, there are no straightforward ways to extend relational queries for spatial queries, including PIP test based spatial joins.

In the research community, there are increasing interests in using GPGPU technologies for data management although we are not aware of the existence of such commercial or open source products from the market yet. Two pioneering works, i.e., GDB [7] from HKUST and Sphyraena [8] from University of Virginia, have investigated the potentials of using GPUs for managing relational data. Sphyraena has provided a SQL interface based on SQLLite[3], however, its functionality is limited to mostly selection types of queries. GDB has more support for join-related queries and several indexing modules have been provided to speed up relational join processing. More recently, a more complete set of relational algebra algorithms have been implemented by a group of researchers at the Georgia Tech University [9] and reportedly they have achieved better performance on new generations of Nvidia GPUs. Similar to relational data management on multi-core CPUs, it is unclear how the parallel relational data management and query processing techniques can be extended to geo-referenced spatial data that has quite unique operations, for example PIP test.

Due to the close relationship between SDB/GIS and Computer Graphics and Design Automation that also handle spatial data, it is natural to adapt exiting indexing approaches for vector geospatial data, including both point and polygon data. Although the GPU-based R-Tree construction and query processing [10] is general enough to be applied in our applications (but with some limitations as detailed next), many of the spatial indexing approaches [11-14] designed for computer graphics applications are not suitable for database applications. While these indexing structures are useful for ray-tracing and iso-surface constructions, they are not suitable for spatially joining multiple datasets that is commonly required in SDB/GIS.

There are several independent works that are related to our efforts in building a GPU-based, high-performance system to perform geospatial queries although they might initially targeted at different application areas. While there are several attempts to implement the classic R-Tree spatial indexing structure [2, 3] on GPUs, the work reported in [10] seems to be the most comprehensive one. The authors have tested parallel spatial range queries on built 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 [1, 14-17], we are skeptical on whether R-Trees are good choices for spatial joins on GPUs as data accesses are highly irregular when pairing the bounding boxes of geometrical objects indexed by R-Trees in the source and target datasets to be joined. The problem has also been observed in a research on all-pair nearest neighbor queries on CPUs [18].

The in-memory grid files data structure on GPUs proposed by [19] is closely related to the simple grid file structure we have used for the filtering phase of the spatial join as both of them are derived from classic grid file structures. However, there are several key differences between the two. First of all, their grid file is designed to process individual queries while our grid file is designed to process spatial join. Second, the grid file in [19] 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 finds unique pairs of point quadrants and polygons which requires complex post-processing including sorting, searching and removing duplicates.

Finally, there are a few parallel spatial join algorithms proposed in the past few decades that mostly targeted at shared-nothing parallel hardware architectures [20-25]. Many of these techniques rely on data declustering techniques to reduce slow disk I/Os and network communication costs. More recently, some of the spatial join algorithms have been adapted to the MapReduce parallel computing environment [26, 27]. However, parallel data processing with MapReduce has been criticized for its inherent limitations on performance and efficiency [28]. In contrast, in this study, we aim at designing and implementing spatial joins on modern commodity GPUs which closely resemble supercomputers as both implement the primary Parallel Random Access Machine (PRAM[4]) characteristic of utilizing a very large number of threads with uniform memory latency (such as Cray XMT[5]) [29]. We are not aware of previous research on parallel spatial join on GPUs other than similarity join on point data [30].

[2] http://www.opengeospatial.org/standards/sfs
[3] http://www.sqlite.org/
[4] http://en.wikipedia.org/wiki/Parallel_Random_Access_Machine
[5] http://www.cray.com/products/XMT.aspx

3. SYSTEM IMPLMENTATION

3.1 Overview

The workflow of the proposed approach is illustrated in Fig. 1. We first divide points into quadrants so that each quadrant contains at most K points. The bounding boxes of the points within quadrants are then computed. In order for a point to be inside a polygon, the bounding box of the quadrant that the point falls within and the bounding box of the polygon must intersect. We thus can filter out point quadrant and polygon pairs whose bounding boxes do not intersect and only perform PIP tests for points in quadrants whose bounding boxes intersect with the bounding boxes of one or more polygons. The output of the filtering phase is a list of quadrant identifiers and each quadrant identifier is associated with a list of polygon identifiers representing the intersected polygon bounding boxes. For an element in the output list, i.e., a (qid, {pid}) pair, we assign the pair to a GPU computing block. Within the computing block, each thread is responsible for processing a single point and test whether the point is within any of the polygons represented by the set of polygon identifiers. As the polygons representing the real world zones are usually mutually excusive (i.e., spatially non-overlapping), when a point is determined to be within a polygon (to be detailed in Section 3.5), the PIP test for the point is terminated.

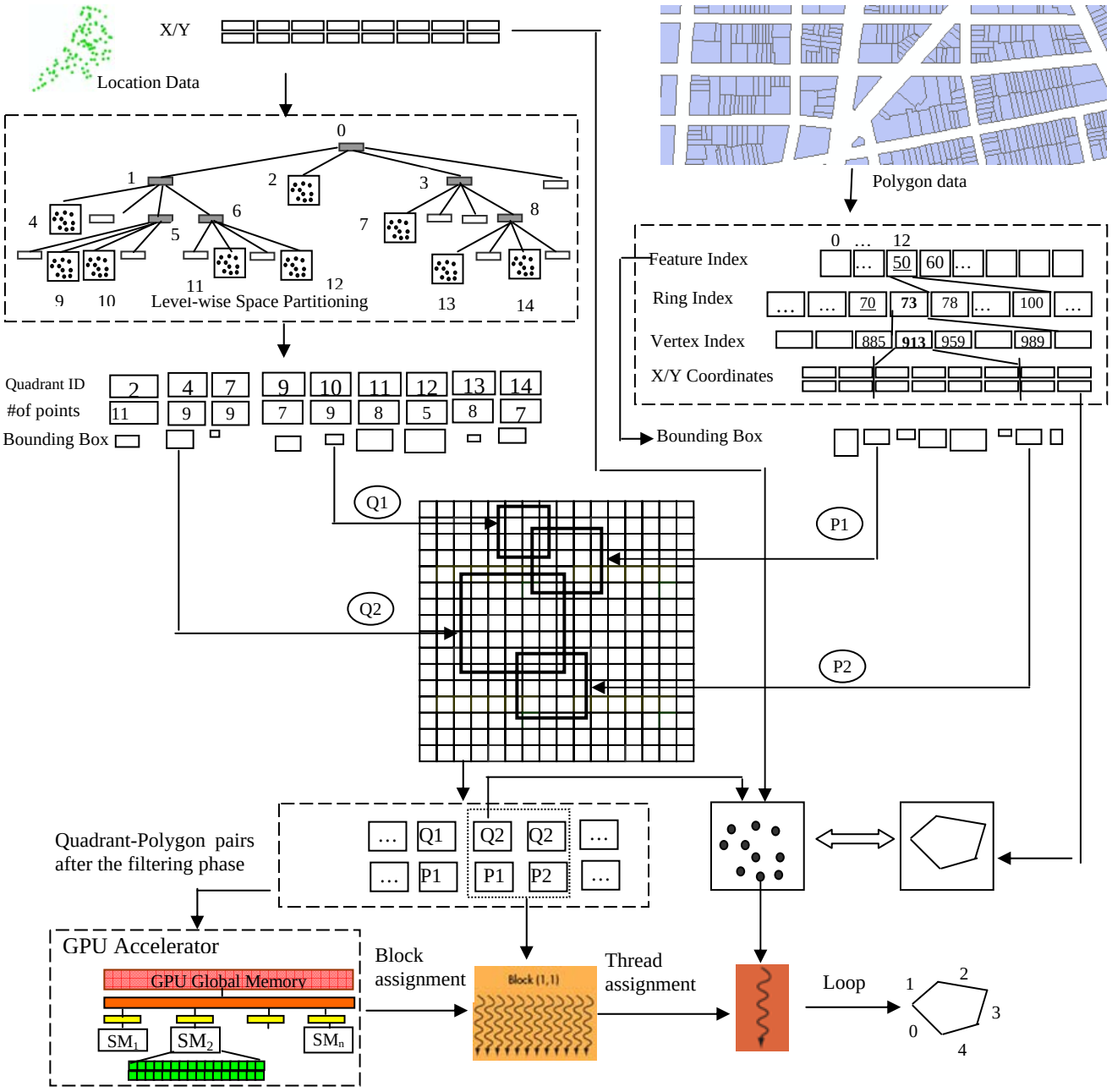

Fig. 1 Framework of Spatial Join on Points and Polygons using PIP Test on GPU

As illustrated in Fig. 1, the point coordinates and the bounding boxes of quadrants are laid out as one-dimensional arrays so that they can be easily streamed among hard drives, CPU memories and GPU memories. The polygon data is laid out by following the same design principle although the data layout is a little more complex. While the design details of the data layouts is deferred to Section 3.2, we would like to note that these simple in-memory data structures are quite effective in reducing I/Os (which are getting increasingly expensive on modern hardware) and play an important rule in achieving the desired speedup. The implementation details on generating point quadrants, computing the (qid, {pid}) list and performing PIP tests on GPUs are provided after introducing the in-memory data structures in the next few subsections. We note that while we strive to optimize our implementation to achieve good performance, our existing implementation is largely built on top of the Thrust parallel primitives[6] which may not have the best achievable performance due to the limitations on using parallel primitives. It is well known that there is a tradeoff between coding complexity and code efficiency in using such parallel libraries and we believe there are still quite some rooms for future improvements with respect to performance. Nevertheless, the implementation can serve as a baseline to understand the potentials of GPU accelerations in spatial databases and GIS.

3.2 In-Memory Data Structures

The use of in-memory data structures to store the point coordinates and polygon vertices was actually forced by the CUDA computing model that favors one-dimensional arrays, especially in the previous versions of compute capability. As such, these one-dimensional arrays are used extensively to store point and polygon data in both CPU memory and hard drives as well for easy data streaming among them. Although traditional SDB and GIS implementations favor dynamic memory allocations and use pointers extensively, we have found that using the simple array-based in-memory data structures improves system performance considerably due to its cache friendly property. This has motivated us to seek a systematic way to layout large-scale point data and polygon data to balance between performance and memory footprint. Fortunately, many real world point data have meaningful spatial and temporal granularities which can be used to segment the point data into chunks that are small enough to fit into CPU/GPU memory and big enough to maximize disk I/O performance. The point locations that we use in this study actually are the pickup and drop-off locations of taxi trip records in NYC. On average there are about half a million taxi trips per day. Assuming each point takes 8 bytes for lat/lon coordinates, the chunk for a month would be around 120 MB per month which seems to be a good choice given that the machine we use for the experiments has 32 MB hard drive cache and the CPU and GPU memory capacities are 16 GB and 6 GB, respectively.

Unlike the point data that can be stored as one-dimensional arrays in a straightforward manner, auxiliary information is needed to store polygons as arrays. This is because polygons have variable numbers of vertices and a polygon may have multiple rings, e.g., polygons with holes. Since OGC SFS has been widely adopted by the SDB and GIS communities, our in-memory data structures for polygons are designed to support the standard. According to the specification, a polygonal feature may have multiple rings and each ring consists of multiple vertices. As such, we can form a four level hierarchy from a data collection to vertices, i.e., dataset → feature → ring→ vertex. Five arrays are used for a large polygon collection. Besides the x and y coordinate arrays, three auxiliary arrays are used to maintain the position boundaries of the aforementioned hierarchy. As shown in the top-right part of Fig. 1, given a dataset ID (0..N-1), the starting position and the ending position of features in the dataset can be looked up in the feature index array. For a feature within a dataset, the starting position and the ending position of rings in the feature can be looked up in the ring index array. Similarly, for a ring within a feature, the starting position and the ending position of vertices belong to the ring can be looked up in the vertex index array. Finally, the coordinates of the ring can be retrieved by accessing the x and y coordinate arrays. It is easy to see that retrieving coordinates of single or a range of datasets, features and rings can all be done by scanning the five arrays in a cache friendly manner. It is also clear that the number of features in a dataset, the number of rings in a feature and the number of vertices in a ring can be easily calculated by subtracting two neighboring positions in the respective index array. As such, the array representation is also space efficient.

To convert existing disk-resident polygon data in various formats into the array based representation, we use an open source software called GDAL[7] to access polygon datasets, polygons, rings and vertices sequentially and output polygon vertices and indexing positions to the respective arrays in a way similar to ETL (Extract, Transform and Load) in relational databases and data warehouses [8]. While this step is usually I/O intensive due to frequent disk accesses and extensive dynamic memory allocation and de-allocation to accommodate variable-sized polygons, this is a one-time process and the resulting arrays can be written to hard drives and streamed to CPU memories afterwards. Although a polygon may have a large number of vertices in practice, the number of polygons is relatively small and the volume of the polygon data (including the auxiliary indices) is far less than point location data. For the NYC census block dataset which we use in the experiments, the number of polygons is in the order of 40 thousand and the number of vertices is in the order of 5 million which can be fit in both CPU and GPU memories easily. In contrast, there are nearly 170 million pickup and drop-off locations and the memory footprint may already be out of the capacity of some GPU devices. Note that many GPU operations require intermediate storage which will further reduce the number of points that can be processed in a single run. The algorithms to be presented in the subsequent three subsections have taken the device memory constraints into consideration and allow processing hundreds of millions of point locations.

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

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

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

3.3 Generating Point Quadrants

Many real world point locations are clustered and neighboring points often behave similarly. For example, tourist attractions often receive a large number of taxi pickups and drop-offs. The locations are usually close to each other and are associated with a same zone. To group the large number of points into chunks for parallel spatial join, we have developed an approach to hierarchically divide the point data space into quadrants and identify quadrants that have fewer than K points. Quadrants that have more than K points are further divided using the same principle until either all points are grouped or the maximum level (M) is reached. The process is similar to quadtree constructions [2, 3] but our approach adopts a top-down subdivision strategy and can be efficiently implemented using GPU-based parallel primitives provided by the Thrust library which is now part of the CUDA SDK. Compared to native CUDA programming which usually have a deep learning curve in order to achieve high efficiency, parallel primitives provide a nice tradeoff between coding complexity and code efficiency. While it is beyond our scope to present the details of primitives based parallel programming, the appendix provides a brief introduction to several parallel primitives that are needed in generating point quadrants from large-scale point locations in parallel on GPGPUs.

The procedure of generating point quadrants is presented in Fig. 2 where the names of parallel primitives are bolded and underlined and the variables (either a vector or a scalar) names are bolded and italicized for easy interpretation. An illustrative example is also provided in Fig. 3. Steps 1-3 in Fig. 2 are used to sort points (stored in P) based on their level k Morton codes (used as keys, stored in PK) and count the numbers of points (stored in UN) associated with the unique keys (stored in UK). The points are also sorted based on the keys so that they can be reordered later (Steps 7 and 8) and get ready for the next level. Steps 4 and 5 are used to identify quadrants and the points associated with the quadrants. Note that the SIGN vector indicates whether a quadrant is identified and the INDICATOR vector indicates whether a point belongs to an identified quadrant. For each number in UN (assuming *n*), which records the number of points with the same level-k key, the boolean value in the SIGN vector at the same position will be replicated *n* times in INDICATOR. This is done by using the Expand parallel primitive that has been implemented by combining a *Scatter* and a *Gather* primitive (to be detailed next).

Inputs: vector of point dataset ***P***

Outputs: re-arranged point dataset ***P,*** quadrant key vector ***LK,*** vector of numbers of points falling within quadrants ***LN,*** vector of numbers of starting positions of points in quadrants ***PN,*** number of quadrants ***n_l*** and number of points falling within the quadrants ***n_s*** **(**at all levels)

Initialization: Set ***n_p*** (representing number of identified points in resulting quadrants) to 0 and set ***n_q*** (representing number of identified quadrants) to 0.

For ***k*** from 1..***M*** levels (with starting quadrant at **n_q** and starting points at **n_p**):

1. **Transform** point dataset ***P*** to key set ***PK*** using Z-ordering at level **k.**
2. **Sort_by_key** using ***PK*** as the key and ***P*** as the value
3. **Reduce_by_key** using PK as the key and copy the unique keys to ***UK*** and numbers of the same key in each key group into ***UN***
4. Classify each quadrant (corresponds to a key in ***UK***) based on whether the numbers in ***UN*** is above (set to 0) or below (se to 1) the threshold **K** and copy the result to a boolean vector ***SIGN*** by using **Transform**.
5. Identify points that are within or not within the quadrants to be pruned based on ***UN*** and ***SIGN*** by using **Expand** and output the result to a boolean vector ***INDICATOR***.
6. Copy the identified quadrant keys to ***LK*** and number of points in the quadrant to ***LN*** by using ***Copy_if*** based on ***UK, UN*** and ***K;*** also set ***n_l*** to the number of identified quadrants at the level and **n_s** to the number of points fall within the quadrant
7. **Copy** all points in **P** that are in the identified quadrant to **PL** and those that are not in the identified quadrant to ***PQ*** using **Copy_if** based on ***INDICATOR***
8. Combine ***PL*** and ***PQ*** to **P**using **Copy** by placing ***PL*** ahead of ***PQ***
9. Keep elements in ***PK*** correspond to points that fall within the identified quadrants and remove the rest using **Remove_if**
10. Increase ***n_p*** by ***n_s*** and increase ***n_q*** by ***n_l.***

Process points that fall within the last level quadrants but have more than K points

11. **Transform** point dataset ***P*** to key set ***PK*** using Z-ordering at level **k** starting at **n_p** (similar to Step 1)
12. **Reduce_by_key** using PK as the key starting at **n_q** and copy the unique keys to ***LK*** and numbers of the same key in each key group into ***LN*** starting at **n_q** (similar to step 3)

Compute bounding boxes for quadrants

13. Transform point dataset ***P*** into bounding boxes ***B*** using **Transform**
14. **Reduce_by_key** on ***B*** using ***PK*** and store the result in ***QB***

Fig. 2 Algorithm of Parallel Primitives Based Level-Wise Point Quadrant Generation

To better illustrate how the parallel primitives work together to identify tree leaf nodes, let us consider the following SQL statement "SELECT * FROM T WHERE #key IN (SELECT #key FROM T GROUP BY #key HAVING COUNT (#key)) > #K". The statement selects individual tuples that satisfy a count-based group condition and does what we want in Step 5. While it is straightforward to output individual tuples whose #key values are in the resulting single-attribute relation of the sub-query with the group by/having clauses on CPUs, it is neither convenient nor efficient to perform set membership tests on GPUs. Actually we do not have to due to the relationships among UK, UN and PK. Obviously UK and UN are group-related (when referencing to the SQL statement). The evaluation results of the having condition should be a boolean vector (i.e., SIGN) that has the same length as UK and UN. Since UK and PK has the same key order, when mapping UK back to PK, each

boolean value at SIGN[i] will repeat exactly UN[i] times and the vector of such boolean values (INDICATOR) exactly indicates whether a key in vector PK satisfy the group-based criteria (i.e., the condition specified in the having clause in the example SQL statement). Now the problem translates into how to generate the INDICATOR boolean vector from the SIGN boolean vector and the UN integer vector. This actually can be done using four parallel primitives that are introduced in the Appendix. First, an exclusive scan is performed on UN to compute the group boundaries. Second, a Scatter primitive is used to scatter the group boundary values to proper positions in a temporal vector (VT) that has the same size as PK/INDICATOR. Third, a Scan primitive using the max associative function is performed to propagate the boundary values in VT to positions within group boundaries. Finally, a *Gather* primitive is applied to update the values in INDICATOR by the values in SIGN using VT as the map, i.e., the i^{th} element in INDICATOR is assigned to the value of SIGN[VT[i]].

Step 6 copies the identified quadrant and the corresponding numbers of points to two new vectors (LK and LN. This step also computes the number of identified quadrants and the number of points that fall within the quadrants in order to set the proper level boundaries in step 10. Steps 7 and 8 actually rearrange the points by moving points in identified quadrants to the left and the rest of the points to the right so that the next level only needs to process the non-identified points. Step 9 removes elements in PK that correspond to points that do not fall within any of the identified quadrants at the level. It is possible that some last-level quadrants have more than K points and they can not be identified in steps 1-10. As such, step 11 (similar to step 1) and step 12 (similar to step 3) are used to process these points which can be considered as a simplification of steps 1-10 since no sorting and reordering are needed at the last level.

After the quadrants are identified, computing the bounding boxes of the points that fall within the quadrant becomes embarrassingly parallelizable by using a transform primitive and a reduce_by_key primitive. The transform primitive converts a point into a bounding box by setting the top-left and bottom-right points of the bounding box to the point itself (Step 13). Since there is a one-to-one correspondence between PK and P and the unique values of PK and the identified quadrants, a straightforward reduce_by_key primitive can be applied to compute the bounding boxes of points in the identified quadrants by using a user-defined functor that takes the extremes of the two bounding boxes and form a larger bounding box. We note that by applying an exclusive scan on LN we can obtain the starting positions of the first points of the identified quadrants in P and hence the point coordinates can be accessed in parallel for subsequent PIP test.

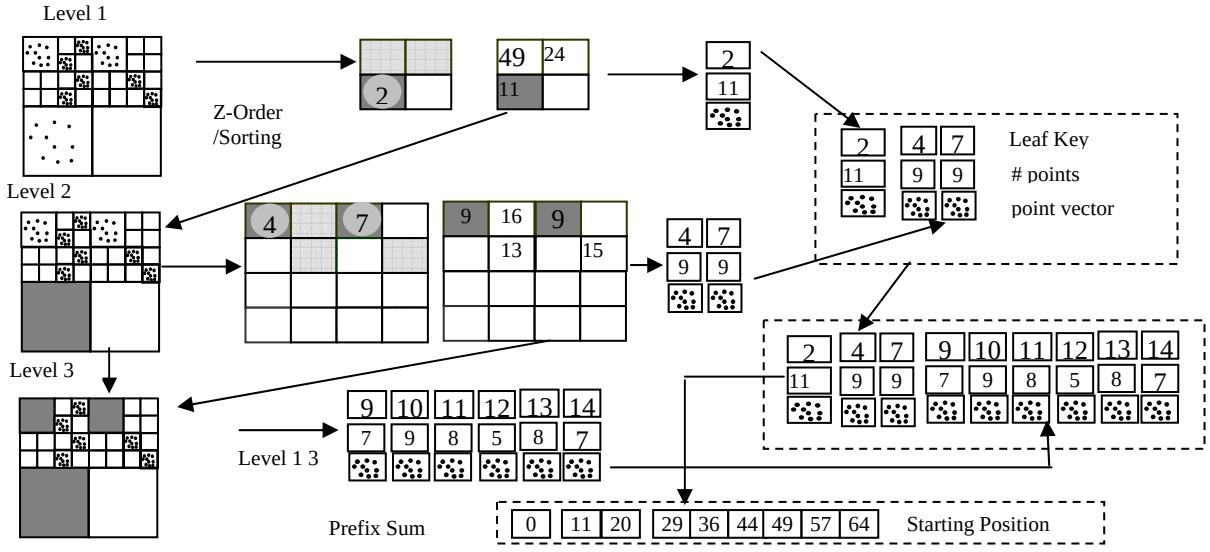

Fig. 3 A Running Example to Illustrate the Process of Generating Point Quadrants using Parallel Primitives

3.4 Associating Quadrants and Polygon Bounding Boxes with Grid Cells

Pairing all point quadrants with all polygons for PIP test is computationally prohibitive for even relatively small number of quadrants and number of polygons. This is not necessary either as the majority of the point quadrants will only intersect with a small number of polygons. Quite a few spatial indexing approaches have been proposed for filtering purposes but the majority is serial in nature and hence is not suitable for GPU implementation. We propose to use a simple grid file spatial indexing structure to index both the bounding boxes of point quadrants and the bounding boxes of polygons. Our approach converts the problem of comparing spatial relationships (e.g., intersection of bounding boxes) in spatial queries into a searching problem which can be efficiently implemented by integrating an in-house developed kernel with the vectorized binary search parallel primitive that is recently supported by the Thrust library in the CUDA SDK.

As illustrated in Fig. 4, we first rasterize the bounding boxes of both the point quadrants and polygons using a uniform grid. Given a fixed grid cell size, the number of rows and the number of columns of the bounding boxes to be rasterized can be easily computed. If two bounding boxes overlap then they will have at least one common grid cell. To find all the polygons that a point quadrant intersects, for each grid cell of the bounding box of the point quadrant, we search the grid cell identifier in the rasterized grid cells of the bounding boxes of polygons. If there is match, then we pair the point quadrant and the polygon for further refinement in the next stage. Note that as a bounding box of a point quadrant usually has multiple grid cells and each grid cell is searched and matched independently, there will be (potentially a large number of) duplicates of the pairs of point quadrants and polygons. These duplicates need to be removed before the refinement phase. We have implemented this procedure by using a combination of binary_search and lower_bound primitives provided by Thrust. We refer to the appendix and Thrust documentation for details on these two parallel primitives In the example show in Fig. 4, among the 12 cells of Q1, two cells have successfully the corresponding cells in the rasterized cells of the polygon bounding boxes (and we term it as "paired"). Similarly one cell in Q2 is paired with one cell in P1 and two cells in Q2 are paired with two cells in P2. After applying the unique primitive (see appendix) we will obtain three pairs. After applying the sort primitive, we get two pairs (c.f. Section 3.1), i.e., (Q1,{P1}) and (Q2, {P1,P2}) and they are ready to be sent to GPU computing blocks for PIP tests.

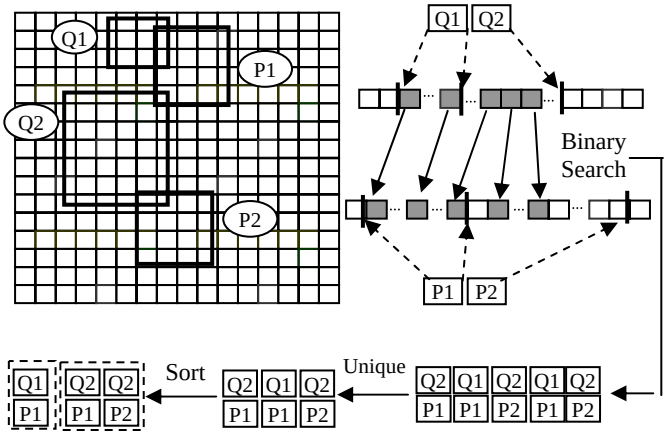

Fig. 4 Illustration of Grid-File based Spatial Join Filtering and its GPU implementation using Parallel Primitives

While quite a few parallel primitives that are needed in this step have been provided by the Thrust library, such as vectorized binary search, unique and sort, we have found that it is difficult to rasterize bounding boxes using existing parallel primitives although it seems to be straightforward to split a rectangle into a set of squares and assign an identifier to each of the cells. As such, we have developed a simple CUDA kernel for this purpose. Before launching the rasterization kernel, we apply an exclusive scan kernel to compute the starting positions of where each rasterized bounding box should begin to write out their grid cell identifiers on the vector of the total number of cells of the respective bounding boxes. The numbers can be easily calculated as the multiplications of the numbers of rows the numbers of columns of the respective bounding boxes. With all these input information, each thread is assigned to process a bounding box in parallel and write out the cell identifiers sequentially.

3.5 Parallel PIP Test

As discussed previously (c.f. Section 3.1 and Fig. 1), each (qid, {pid}) pair is assigned to a computing block to utilize the first level parallelism in GPGPU computing. Within a computing block, there are quite some parameters to be fine-tuned to make full use of the fine-grained thread level parallelism on GPUs, such as determining number of threads per block, decisions on using shared memory and approaches to mitigate register variable pressure. In this subsection, we report our design and implementation of the parallel PIP test for all points in a quadrant and all the candidate polygons that are derived by the filtering phase.

There are quite a few computational geometry algorithms for point in polygon test and we refer to [31] for a brief review. While PIP test algorithms that require preprocessing may obtain sub-linear complexity, algorithms that do not require preprocessing are usually simpler and more suitable for GPU implementation. In this study, we use the most popular ray crossing (ray intersection) algorithm with a complexity of O(n) where n is the number of edges of a polygon. The basic idea of the ray crossing algorithm is illustrated in Fig. 5. If a ray emanating from a test point crosses the boundary of a polygon odd times, it is inside a polygon otherwise outside a polygon. In this study, we have adopted the concise code provided by Randolph Franklin (listed in Fig. 5) and modified it to run on GPUs. We note that the open source GIS packages Java Topology Suit (JTS)[9] and its C/C++ translation Geometry Engine - Open Source (GEOS)[10] also have implemented a similar ray crossing algorithm for PIP test. GEOS has been integrated into PostGIS[11]/PostgreSQL[12] to support PIP test in the form of the ST_Intersects function when the two inputs are point and polygon geometry objects, respectively. As such, it is fair to compare a parallel GPU implementation with a serial CPU implementation of the same algorithm.

We have adopted a simple thread level parallelization schema within a computing block, i.e., each thread is responsible for testing whether it is within the paired polygons. The simple design has two advantages with respect to GPU device memory access. First, as points are stored consecutively in their coordinate arrays (c.f. Section 3.2), neighboring threads will access consecutive memory addresses which is coalesced perfectly. Second, all threads will access the same polygon vertices which are also stored consecutively in their coordinate arrays. The GPU hardware is able to broadcast the requested vertex coordinates to all the requesting threads which significantly reduces memory access costs. While originally we had planned to use GPU shared memory to store both coordinates of points in a quadrant and polygon vertices, as GPU device memory accesses are already optimized, using shared memory actually decreases performance due to

[9] http://www.vividsolutions.com/jts/jtshome.htm
[10] http://geos.osgeo.org
[11] http://postgis.refractions.net/
[12] http://www.postgresql.org/

synchronization costs which is necessary after having threads collaboratively load data from device memory to shared memory. Not using shared memory will also improve scalability as the numbers of points and polygons that can be assigned to a computing block is not limited by shared memory sizes any more. If the number of points assigned to a computing block exceeds the number of threads that is allowed by a computing block, the points can be divided into chunks and simply have threads loop through the chunks.

```
int pnpoly(int npol, float *xp, float *yp, float x, float y)
{
    int i, j, c = 0;
    for (i = 0, j = npol-1; i < npol; j = i++) {
      if ((((yp[i] <= y) && (y < yp[j])) ||
           ((yp[j] <= y) && (y < yp[i]))) &&
          (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i]))
        c = !c;
    }
    return c;
}
```

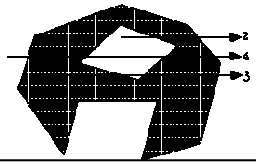

Fig. 5 Illustration and Code Segment of Ray Crossing based Point-in-Polygon Test (see [13]for more details)

While shared memory is not a limiting factor in our design and implementation, we have found that the limited number of register files available to a thread becomes a bottleneck. Due to the complexity of the algorithm, we can not reduce the number of registers used by a thread to below 44, which is more than the number of registers allowed when a SM is fully utilized under CUDA compute capability 2.0 (32768/1024=32). As reported in the experiment section, we have found that using 256 threads per block seems to achieve the best performance for a quadrant size K=512. However, the occupancy in this case is only 33% which is far from optimal due to the register file limit under compute capability 2.0. We expect the occupancy will be improved under compute capability 3.0 where each thread is allowed to use 65536/1024=64 register variables when a SM is fully utilized. Another option to try is to allow register spilling which requires more careful design to improve the overall performance.

4. EXPERIMENTS

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 in 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 Census 2000 dataset[14]. There are more than 40 thousand census block polygons in NYC with more than 5 million vertices. All experiments are performed on a Dell Precision T5400 workstation equipped with dual quadcore CPUs running at 2.26 GHZ with16 GB memory, a 500G hard drive and an Nvidia Quadra 6000 GPU device. 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. We have set M, the maximum number of levels in point quadrant generation, to 8 and each parent quadrant has $2^l*2^l=4*4=16$ child quadrants with l=2. With a cell size of 2 feet at the finest level, the whole NYC area is rasterized into a 2^s*2^s grid where s=l*M=16. As such, all quadrants can be identified by a 32-bit Morton code.

For comparison purposes, we have implemented the same spatial join using open source GIS packages, i.e., libspatialindex[15] to index polygon data by building an R-Tree, and, GDAL, which implicitly uses GEOS, to perform PIP test. 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 and is beyond the scope of this research. As such, the CPU implementation queries each point against the indexed polygons. If the point falls within any of the bounding boxes of polygons, the polygon identifiers will be returned for refinement. It is clear that, while the polygons do not spatially overlap, their bounding boxes can overlap and a point query may return multiple polygons for PIP test in the refinement phase. The CPU implementation performs the PIP test for each of the polygons in the query result set and breaks if any of the PIP test returns true.

Our GPU implementation has several 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 this section, we will first report the best results in Section 4.2. We then provide the experiment results on generating point quadrants in Section 4.3 as it is the most time consuming component in the whole process. In Section 4.4, we vary both N and K and report their impacts on the overall performance.

4.2 Overall Experiment Results

The best performance of our GPU implementation is achieved with K=512 and N=256 with a total end-to-end runtime of 11.165 seconds. In contrast, the serial CPU implementation takes 54,819 seconds (15.223 hours). As such, a significant speedup of 4,910X has been achieved. Note that we have not included the disk I/O times to load the points and polygons as this is one-time cost and is not directly related to the spatial join. Furthermore, as discussed before, these data are stored as binary files on disk. With a sustainable disk I/O speed of 100 MB per second, the point and polygon data can be streamed into CPU main memory in about 15 seconds. Since the disk I/O time is comparable to the spatial join time, even if the disk I/O times are included, the order of speedup will not be changed.

[13] http://local.wasp.uwa.edu.au/~pbourke/geometry/insidepoly/

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

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

We attribute the 3-4 orders of improvements to the following factors. First, all the point, polygon and auxiliary data are memory resident in our GPU implementation. In contrast, the open source GIS packages are designed to be disk resident and data and indices are brought to CPU memory dynamically. While the sophisticated design is necessary for old generations of hardware with very limited CPU memory, current commodity computers typically have tens of gigabytes of CPU memory which renders the sophisticated design inefficient. We also have observed that the open source packages use dynamic memory and pointers extensively which can result in signficant cache and TLB misses. Second, in our GPU implementation, we have divided points into quadrants before we query against the polygons in the filtering phase using a GPU based grid file indexing structure. In the serial CPU implementation, each point queries against the polygon dataset individually. While the polygon dataset is indexed, each point query needs to travel from the root of R-Tree of the polygon dataset to leaf nodes which is quite costly. While not tested in this study, we expect querying the bounding boxes of points in quadrants, instead of querying the points individually, can potentially improve the serial CPU implementation. Third, in addition to the improved floating point computation on GPUs, the massively data parallel GPU computing power is utilized for all phases of the spatial join process, including generating point quadrants, filtering quadrant-polygon pairs and PIP test in computing blocks. While we have not yet been able to separate the contributions of the three factors, we plan to do so by hybridizing CPU and GPU implementations in our future work and measure the performance of different combinations.

Before we provide more detailed results in the next two subsection, we would like to report that among the 11.165 seconds end-to-end runtime for the point-to-polygon spatial join, the majority (8.170 seconds) is spent on generating point quadrants. The runtimes to rasterize bounding boxes of point quadrants and polygons are 0.469 and 0.640 second, respectively. The binary search and duplication removal take 0.265 and 0.405 second, respectively. The PIP test kernel takes 1.206 second which is only 10.8% of the total runtime. We note that the total runtime includes the data transfer time from CPU to GPU for both the point and polygon data, which is 1.030 second.

4.3 Results on Generating Point Quadrants

As discussed in Section 3.3, the modules in generating point quadrants are implemented on top of the Thrust parallel library. Each of the steps listed in Fig. 2 (except Step 10) corresponds to a call to the respective parallel primitive. Since this is the most time consuming step in the whole process, to better understand the distributions of computing workloads, the runtimes of the step groups are listed in Table 1. We have not included the runtimes for processing last level (steps 11/12) as they are negligibly small.

Table 1 Runtimes of Steps in Generating Point Quadrants of 8 Levels (milliseconds)

Quadrant Level	1	2	3	4	5	6	7	8
1 Point Transformation time (Step 1)	26.14	25.65	34.77	34.72	34.67	33.75	27.19	2.23
2 Point Sorting (Key-Value) time (Step 2)	281.80	336.12	494.54	514.75	513.23	546.00	526.84	47.35
3 Key Reducing Time (Step 3)	71.83	71.87	73.24	72.96	73.62	74.36	63.37	6.44
4 Quadrant Identifying Time (Steps 4, 5, 6)	95.27	95.50	94.53	95.98	94.08	93.30	78.39	8.99
5 Point shuffling Time (Steps 7, 8, 9)	246.48	248.17	259.06	263.09	261.98	269.61	271.70	26.05
6 # of identified points	0	15955	291,342	1,351,989	3,546,982	30,510,995	120,775,838	11,551,455
7 # of identified quadrants	0	73	1,938	12,350	36,653	225081	1,250,597	219,730
8 Bounding box derivation time (Steps 13/14)								651.78

From the results we can see that nearly half of the point quadrant generation time is spent on sorting the points with the Morton code as the key and the coordinate as the value. Given that 168 million points are sorted in 3260.63 milliseconds in 8 rounds (levels), we have achieved an overall key-value sorting rates at the 51.52 million pairs per second on the Quadro 6000 GPU device. While the achieved sorting speed seems to be much lower than the previous results of sorting 222 million pairs per second for 32 bit key and 64 bit value on an Nvidia C2050 device with ECC enabled [32], we argue that the actual sorting speed is higher than what they have achieved. Instead of using n directly, which is the total number of input location points, the total number of points that participate in sorting at all the levels is $n' = \sum_{i=1}^{M+1}(n - \sum_{i=1}^{M} n_{i-1})$ where n_i is the number of points that fall within the quadrants identified at the level i (assuming n_0=0). When we plugin the n_is provided at the 6th row of table 1, n' becomes 1,136,417,142 which gives a sorting speed of 348 million pairs per second and is 58% faster than the results reported in [32]. The reason is that keys that are sorted at the previous levels will incur no data movement in the sorting of the next levels due to the nature of radix sort. We speculate that when a customized sorting algorithm is developed for our specific application to allow picking up the sorting results from a previous level without repeating the radix-sort from scratch, the overall sorting performance can potentially be significantly sped up. This is left for our future work.

The next most time consuming component is to move points fall within the identified quadrants ahead of the points whose quadrants are yet to be identified, i.e., point shuffling. From the runtimes shown in the 5th row of Table 1 we can see that the runtimes remain constant until the last level where almost all points have identified their quadrants. Again, we believe this step can be combined with the sorting step when a customized CUDA kernel is developed. As a comment, while using parallel primitives facilitate fast prototyping, implementations built on top of parallel primitives often are not

the ones that can achieve the highest efficiency. In this particular case, the intermediate results in the parallel primitives are not available to the subsequent steps and some duplicated computing can not be avoided.

From the 6th and 7th rows of Table 1, we can also see that the majority of the quadrants/points are identified at level 6 and 7 where the quadrant sizes are 8*8 and 4*4 feet, respectively. This is understandable as the majority of taxi pickup locations are clustered at the street intersections, especially in the midtown and downtown Manhattan area. For the quadrants identified at the higher levels, they will have larger bounding boxes and are likely to intersect with more polygons based on our filtering algorithm. Using a smaller K will reduce the number of such big quadrants to improve the pruning power and improve the PIP test performance in the refinement phase. On the other hand, using a small K will increase the number of quadrants. This in turn will increase the time to generate the quadrants due to the fact that more sorting work is needed to put points in smaller quadrants.

4.4 Impacts of Quadrant Point sizes and Threads Numbers

To further discuss the impacts of the quadrant point sizes (K) and threads numbers (N) on the performance, we have varied K from 256 to 1280 while fixing N to 256. We have varied N from 64 to 704 while fixing K to 512. The results are plotted in Fig. 6 and Fig.7, respectively. Note that "query" time in Fig. 6 refers to pairing quadrants and polygons including binary search and duplication removal times as discussed in Section 3.4. The "total" refers to the sum of the "query" time and the PIP test time in the refinement phase. We have not included the quadrant generation time in Fig. 6 as here we are concerned with the tradeoff between the filtering phase and the refinement phase in a spatial join (conceptually the quadrant generation is considered as part of indexing). From Fig. 6 we can see that, as the maximum number of points in quadrants (K) increases, the query times decrease slightly while the PIP test time reaches the minimum at K=512. The query time decreases as K increase can be explained that both the number of point quadrants and the number of the resulting filtered pairs decrease as K increases and thus less computing is needed in the filtering phase ("query"). As discussed previously, a smaller K will result in smaller bounding boxes which may improve the filtering power (fewer false positives) and can potentially reduce the number of PIP test in the refinement phase. On the other hand, a small K will also increase the number of quadrants. As each quadrant is assigned to a computing block, the overhead of launching computing blocks may increase and can potentially overshadow the benefits of the increased filtering power. The results in Fig. 6 show that K=512 provides a good tradeoff.

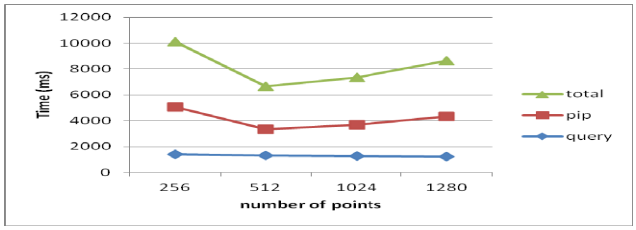

Fig. 6 Variations of Spatial Join Runtimes for Different K Sizes

From Fig. 7 we can see that as the number of threads (N) increase, the runtime for the PIP test in the refinement phase decreases until N=256 before the runtime increases again. Fig 7 also shows a spike of bad performance when N=384. As discussed in Section 3.5, the implementation is limited by the number of register files that can be used by a computing block. When N is small, while a SM can accommodate more blocks (but less than 8 in compute capability 2.0), the threads in a SM is underutilized and the performance is not maximized. As N increases, the occupancy rate gets higher and the performance gets better until N=256 where two computing blocks are accommodated in a SM (three block would require more than 32K registers) and the occupancy rate is 33%. Although increasing N to 352 will increase the occupancy rate to 46%, since 352 does not divide K=512, 160 out of the 352 threads will be idle in the second round of processing the points (c.f. Section 3.5), the performance gets worse. When N is increased to 384, only one computing block can be accommodated in a SM and, similar to N=352, a considerable portion of threads are idle in the second round, the worst performance is observed (the spike in Fig. 7). When N reaches 512 and above, although still only one computing block can be accommodated in a SM, only one loop is needed for the N threads to process the K=512 points and the performance remain the same for N=512 and above. We note that according to CUDA compute capacity 3.0, there will be 64K registers per SM (with the number of maximum thread per computing block remains to be 1024), our implementation can potentially doubles the performance by increasing the occupancy to 66% using N=512.

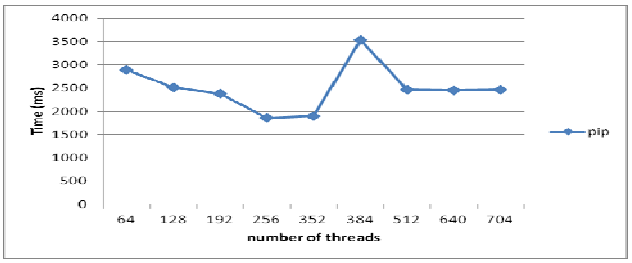

Fig. 7 Variations of PIP Test Runtimes for Different Numbers of Threads in a Computing Block

5. CONCLUSION AND FUTURE WORK

In this study, we have reported our design and implementation on large-scale point-in-polygon test which is a fundamental operation in spatial databases and GIS. The high-performance system can help understand the interactions between people and place more effectively when applied to processing large-scale ubiquitous urban sensing data such as GPS recorded pickup and drop off locations taxi trip records. Experiments have shown that, with a combination of in-memory data structures, algorithm improvement and GPU hardware parallel accelerations, we have achieved 3-4 orders 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 PIP test code). We are optimistic in further reducing the end-to-end runtime. Second, while we have identified factors that have contributed to the significant speedup, the relative contributions remain unclear. We plan to quantify their relative contributions and provide insights on evolving traditional SDB and GIS to modern hardware architectures, including large memory, deep cache hierarchy and parallel processors. Finally, we would like to expand the spatial join framework to other types of spatial joins, such as distance based nearest neighbor.

6. REFERENCES

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

[2] Gaede V. and Gunther O. (1998). Multidimensional access methods. ACM Computing Surveys 30(2), 170-231.

[3] Samet, H. (2005). Foundations of Multidimensional and Metric Data Structures Morgan Kaufmann.

[4] Zhang, J. (2010). Towards personal high-performance geospatial computing (HPC-G): perspectives and a case study. Proceedings of the ACM SIGSPATIAL HPDGIS workshop.

[5] A.Pavlo, E. Paulson et al. (2009). A comparison of approaches to large-scale data analysis. Proceedings of ACM SIGMOD Conference, 165–178.

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

[7] 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).

[8] Bakkum, P. and Skadron, K. (2010). Accelerating SQL database operations on a GPU with CUDA. Proceedings of GPGPU workshop, 94-103.

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

[10] Luo, L., Wong, M. D. F., et al. (2011). Parallel implementation of R-trees on the GPU. Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC).

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

[12] Hou, Q., Sun, X., et al. (2011). Memory-Scalable GPU Spatial Hierarchy Construction. IEEE Transactions on Visualization and Computer Graphics 17(4), 466-474.

[13] Zhou, K., Gong, M., et al. (2011). Data-Parallel Octrees for Surface Reconstruction. IEEE Transactions on Visualization and Computer Graphics 17(5), 669-681.

[14] Andrysco, N. and Tricoche, X. (2011). Implicit and dynamic trees for high performance rendering. Proceedings of Graphics Interface 2011.

[15] Thomas, B., Hans-Peter, K. and Bernhard, S. (1993). Efficient processing of spatial joins using R-trees. Proceedings of ACM SIGMOD Conference.

[16] Huang, Y.-W., Jing, N. and Rundensteiner, E. A. (1997). Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. Proceedings of VLDB Conference.

[17] Papadias, D., Mamoulis, N. and Theodoridis, Y. (1999). Processing and optimization of multiway spatial joins using R-trees. Proceedings of ACM SIGMOD PODS conference.

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

[19] 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.

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

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

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

[23] Patel, J. M. and DeWitt, D. J. (2000). Clone join and shadow join: two parallel spatial join algorithms. Proceedings of ACM GIS Conference.

[24] Kim, J.-D. and Hong, B.-H. (2000). Parallel Spatial Joins Using Grid Files. Proceedings of IEEE International Conference on Parallel and Distributed Systems.

[25] Luo, G., Naughton, J. F. and Ellmann:, C. J. (2002). A Non-Blocking Parallel Spatial Join Algorithm. IEEE ICDE Conference.

[26] 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.

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

[28] 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.

[29] Hong, S., Kim, S. K., et al., 2011. Accelerating CUDA graph algorithms at maximum warp. Proceedings of ACM symposium on PPoPP.

[30] Lieberman, M. D., Sankaranarayanan, J. and Samet, H. (2008). A Fast Similarity Join Algorithm Using Graphics Processing Units. Proceedings of IEEE ICDE Conference.

[31] Zalik, B. and Kolingerova, I. (2001). A cell-based point-in-polygon algorithm suitable for large sets of points. Computers & Geosciences 27(10): 1135-1145.

[32] Merrill, D. and Grimshaw, A. (2011). High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters 21(2): 245-272. Also online at http://code.google.com/p/back40computing/wiki/RadixSorting

Appendix: Parallel Primitives

Although naming conventions might differ slightly under different contexts and software implementations, since our implementation is based on the Thrust library, we next introduce the primitives that we have used in our design using the Thrust terminology.

(1) *Reduce* and *Reduce by key*. *Reduce* is used to simplify a vector/array to a scalar value. For example, reduce([3,2,4])→11. While the summation is frequently used in reductions, Thrust allows using a user defined associative binary function for tailored summation, such as determining the maximum entry or computing bounding boxes of points. *Reduce by key* is a generalization of *Reduce* to key-value pairs based on groups where consecutive keys in the groups are the same. For example, reduce[1,3,3,2],[2,1,3,4])→([1,3,2],[2,4,6]). In this research, *Reduce by key* has been extensively used to compute numbers of points and quadrant that have the same keys based on Morton codes.

(2) *Scan* and *Scan by Key*. The Scan primitive computes the cumulative sum of a vector/array. The Scan primitive can also take a user defined associative binary function. Both the inclusive and exclusive scans are available. For example, exclusive_scan([3,2,4])→([0,3,5]) while inclusive_scan ([3,2,4])→([3,5,9]). Similarly, *Scan by Key* works on consecutive key groups instead of a whole vector/array. In this research, *Scan by Key* is extensively used to compute the positions of entries in a vector after applying *Reduce by key* which outputs numbers of entries with same keys.

(3) *Copy* and *Copy_if*. The functionality of the two primitives is self-evident. In this research, we use *Copy* to move groups of entries from one location to another, mostly within a same vector. The *Copy_if* primitive is mostly used for identifying points and keys (point quadrants) that satisfy certain criteria and output the identified entries to a new vector for further processing.

(3) *Transform*. The basic form of *Transform* applies a unary function to each entry of an input sequence and stores the result in the corresponding position in an output sequence. Transform is more general than Copy as it allows a user defined operation to be applied to entries rather than simply copying. Similar to *Copy_if*, there is also a *Transform_if* primitive which is essentially the combination of *Transform* and *Copy_if*. The combination usually results better performance. In this research, *Transform* has been extensively used to convert points into Morton codes.

(4) *Gather* and *Scatter*. *Gather* copies elements from a source array into a destination range according to a map and *Scatter* copies elements from a source range into an output array according to a map. For example, Gather([3,0,2],[4,7,8,12,15])→([12,4,8]) and Scatter([3,0,2],[12,4,8],[*,*,*,*,*,*])→([4,*,8,*12,*]). Note * values are those unchanged in the third input vector. In this research, we have used the combination of *Gather* and *Scatter* to locate individual points fall within quadrants that have fewer than K points so that they can be moved to proper locations.

(5) *Sort* and *Sort by Key*. *Sort* is probably among the most popular primitives in parallel libraries. In fact, our design aims at utilizing the power of parallel sorting on GPGPUs to speed up generating point quadrants. The current implementations of the sorting algorithms in Thrust are based on a combined radix sort and merge sort which has been proven to be memory bandwidth friendly and practically efficient. Our design facilitates reducing memory traffic and further improves sorting efficiency in the following sense. First, rather than sorting coordinates directly, we sort Z-order transformed Morton codes. The transformation preserves spatial adjacency and requires less data movement. Second, we sort the increasingly longer Morton codes level-by-level and the data movement overheads are amortized among multiple steps since keys and points with the same values do not need to be moved during sorting. Third, keys and points that are identified as those that should be associated with identified quadrants do not need to be sorted any more in the subsequent levels. The last two points have been quantified in Section 4.3. We are also in the process of combining our application semantics and Thrust sorting code to develop a tailored sort primitive implementation to further improve the overall efficacy. This is important as the sort costs are more than half of the end-to-end computing costs in generating point quadrants (see details in Section 4.3).

(6) *Remove_if*. *Remove_if* marks elements in a vector that satisfy a predicate and compact the unmarked elements to the beginning of the vector so that the marked elements are removed. For example, *Remove_if* ([1, 4, 2, 8, 5, 7,is_even])→[1,5,7]. Remove_if is functionally equivalent to *Copy_if* but it allows in-place operation in the Thrust library. In contrast, using *Copy_if* would require a temporary vector and *Remove_if* is more convenient in this case.

(7) *Unique*. *Unique* moves unique elements to the front of a range for each group of consecutive elements. For example, unique([1, 3, 3, 3, 2, 2, 1])→[1,3,2,1]. *Unique* needs to work with *sort* to obtain globally unique elements.

(8) *Binary Search* and *lower_bound*. *Binary Search* searches for values in sorted ranges and needs to work with *sort* for correct searching. When *Binary Search* tells whether the searching elements are in the vector being searched, *lower_bound* tells the position of the searched element. Thrust has provided a vertorized form of both *Binary Search* and *lower_bound.* There is a shuttle implementation issue that requires using *Binary Search* and *lower_bound* together. For example, assuming A=[0,2,5,7,8] and B=[0,1,2,3,8,9], when searching all elements of B in A, conceptually the results should be [0,-1,1,-1,4,-1] where -1 indicates not found. However, *lower_bound* (A,B)→[0,**1**,1,**2**,4,**5**] where the numbers indicate the index of first position where the search value could be inserted without violating the ordering. The numbers are the same as the matching positions of elements if there are matches but meaningless if the searching elements are not in the vector being searched. Fortunately, binary_search(A,B)→[T,F,T,F,T,F] which serves the exact purpose. As such, *Binary Search* and *lower_bound* need to be used together.