High-Performance Spatial Query Processing on Big Taxi Trip Data using GPGPUs

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

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

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

Abstract— City-wide GPS recorded taxi trip data contains rich information for traffic and travel analysis to facilitate transportation planning and urban studies. However, traditional data management techniques are largely incapable of processing big taxi trip data at the scale of hundreds of millions. In this study, we aim at utilizing the General Purpose computing on Graphics Processing Units (GPGPUs) technologies to speed up processing complex spatial queries on big taxi data on inexpensive commodity GPUs. By using the land use types of tax lot polygons as a proxy for trip purposes at the pickup and drop-off locations, we formulate a taxi trip data analysis problem as a large-scale nearest neighbor spatial query problem based on point-to-polygon distance. Experiments on nearly 170 million taxi trips in the New York City (NYC) in 2009 and 735,488 tax lot polygons with 4,698,986 vertices have demonstrated the efficiency of the proposed techniques: the GPU implementations is about 10-20X faster than the host system and complete the spatial query in about a minute. We further discuss several interesting patterns discovered from the query results which warrant further study. The proposed approach can be an interesting alternative to traditional MapReduce/Hadoop based approaches to processing big data with respect to performance and cost.

Keywords- High Performance, Spatial Query, Big Data, Taxi Trip, GPGPU

I. INTRODUCTION

Taxicabs in many cities have been equipped with GPS devices and fare collection systems. Different types of trip related information, such as pickup and drop-off locations and timestamps, fares, trip durations and distances, have been automatically collected for billing and regulation compliance purposes. In the New York City (NYC), more than 13,000 taxicabs generate nearly half a million taxi trip a day on average which amounts to nearly 170 million trips in 2009. These taxi trips, when integrated with urban infrastructures, such as road networks and different types of zones, can be enormously useful for understanding traffic and travel pattern across NYC at different time periods and facilitating urban planning. While there are well established data management techniques, such as Geographical Information Systems (GIS), Spatial Databases (SDB) and Moving Object Databases (MOD), to manage such geo-referenced spatiotemporal data, the huge data volumes have prevented these existing technologies, which are mostly designed for disk-resident systems based on serial algorithms and running on uniprocessors, from achieving performance close to real-time responses to support interactive inquiries. For example, our previous experiments have shown that, simply uploading the raw data of 170 million taxi trip records in a PostgreSQL database and create a geometry column for the pickup locations would cost 100+ hours on a high-end workstation with 48 GB memory and reasonably up-to-date dual Intel Xeon quad-core processors [1]. Although the hardware can potentially provide much higher performance when its parallel processing power and large memory capacity are fully utilized, there is a significant gap between much needed high-performance and the level of achievable performance using existing software stack for big taxi trip data.

While it is certainly desirable to design more sophisticated data structures and efficient algorithms to further improve the efficiency of serial designs and improve the performance of existing technologies, we consider exploiting parallel processing power, which is already economically available on commodity hardware ranging from desktops to workstations to virtual clusters in cloud computing, to be an effective alternative to handling the Big Spatial Data problem [2]. Different from most of the existing studies based on MapReduce/Hadoop techniques to distribute workload to multiple distributed computing nodes to achieve high-performance (e.g., [3,4]), in this study, we aim at utilizing the massively data parallel processing power on commodity Graphics Processing Units (GPUs) for spatial query processing on big taxi trip data and demonstrating its feasibility and efficiency. Our work is built on top of the framework of developing a high-performance data management system for large-scale Origin-Destination (OD) data on modern parallel hardware [1]. While we have addressed point-to-polyline distance type of queries on both multi-core CPUs and many-core GPUs in [1], we will focus on Nearest Neighbor (NN) type of queries between points and polygons in this study. Although both types of queries can be modeled within a general spatial join framework [5] and the yearly 170 million taxi trips in 2009 are used in both studies, the polygon dataset we used in this study has 735,488 tax lot polygons with 4,698,986 vertices, which is much more complex than the street network data we use in [1] with 147,011 polylines and 352,111vertices (5X and 13X, respectively). This brings significant higher computing intensity which makes GPU computing more desirable for analyzing taxi trip data.

The rest of the paper is arranged as the following. Section 2 introduces background, motivation and related work. Section 3 presents the workflow for taxi trip analysis based on spatial query processing. Section 4 provides the design and implementations of GPU-based spatial query processing techniques. Section 5 provides experiment and results on runtimes of system modules and discusses patterns identified from the taxi trip data. Finally Section 6 is the conclusions.

II. Background, Motivation and Related Work

Identifying travel patterns from recorded trips is important to understand human mobility and transportation planning. Existing approaches to trip purpose identification include traditional diary/phone based travel survey and more recently, GPS based travel survey [6,7]. As almost all taxicabs in cities of the developed countries have been equipped with GPS devices and different types of trip related information are recorded. For example, the more than 13,000 GPS-equipped medallion taxicabs in the New York City (NYC) generate nearly half a million taxi trips per day and more than 170 million trips per year serving 300 million passengers. The number of yearly taxi riders is about 1/5 of that of subway riders and 1/3 of that of bus riders in NYC, according to MTA ridership statistics [8]. Taxi trips play important roles in everyday lives of NYC residents (or any major city worldwide) and understanding the trip patterns is instrumental in transportation modeling and planning. However, the large-scale taxi trip data at the level of hundreds of millions are well beyond the processing power of existing GIS and spatial databases technologies. As such, new computing infrastructure that can handle big taxi trip data is needed to process the data and identify trip patterns (such as trip purpose analysis) efficiently and effectively.

As spatial is an important feature for many types of data (especially geo-referenced spatial data that is close to our everyday life) and spatial data volumes are ever increasing, several pioneering works have addressed the scalability issue on processing large-scale geospatial data. In the pre-Hadoop age, parallelization on spatial indexing and spatial join were based on low-level computation protocols which made their adoptions in practical applications very challenging (see [9] for a survey). In recent years, there are significant research and application interests in adopting MapReduce/Hadoop based techniques for geospatial data processing (e.g. [3,4]). The coarse-grained task-level parallelization model makes it relatively easy to adapt existing serial designs on a single CPU core to multiple CPU cores across distributed computing nodes. The availability of cloud computing resources also makes developing and deploying such systems much easier. Despite MapReduce/Hadoop based techniques are generally considered easy to use and have good scalability, they are also criticized for low resource utilizations [10] which makes improving single node efficiency desirable. Indeed, if more data can be processed within a computing node by fully utilizing the increasing number of processing cores and high memory bandwidth (which is typically 3 orders higher than disk I/O speed and 2 orders higher than network bandwidth, i.e., 10-100 GB/s vs. 0.1-1 GB vs. 10-100 MB/s), the intra-node communications and disk I/Os (which is arguably the most expensive part) will be significantly reduced and the overall system performance can be improved. In this study, we aim at improving single-node computing efficiency by making use of GPU computing power.

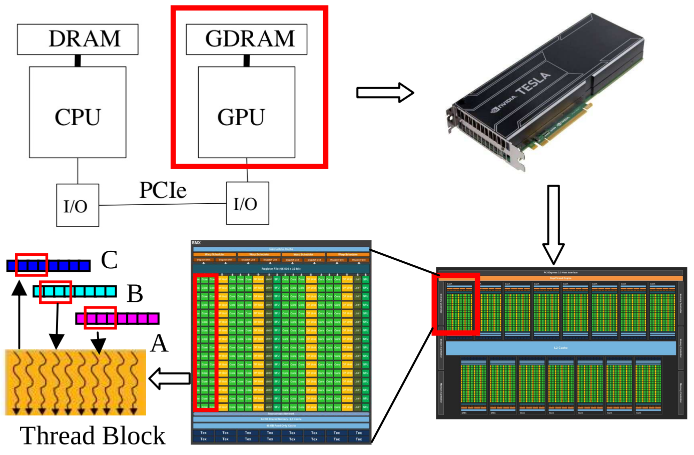

Fig. 1 Illustration of GPU hardware Architecture

Compared with distributed computing and multi-core CPU computing, using GPUs for general purpose computing (or GPGPU in short) is relative young. While we refer to [11] for more details of GPU computing, Fig. 1 illustrates some key features of GPU hardware and Nvidia CUDA (Computing Unified Device Architecture [11]) programming model. Currently, GPUs are typically equipped as PCI-E devices to workstations and computing nodes but have their own graphics memory (top of Fig. 1). While transferring data between CPU memory and GPU memory incur additional overheads, very often offloading computing intensive tasks to GPUs is still beneficial due to their excellent parallel computing power, including large number of processing cores and high memory bandwidth. Roughly speaking, the GPU computing model supports both task parallelism at the thread block level and data parallelism at the thread level (Fig. 1). For a single GPU kernel designed for solving a particular problem, the boundary between task and data parallelism can be configured when the kernel is invoked (lower-left part of Fig. 1). However, to maximize performance, data items should be grouped into basic units that can be processed by a warp of threads (which are dynamically assigned to processor cores) without incurring significant divergence.

While GPUs, as shared-memory parallel hardware, are generally considered lacking good scalability when compared with shared-nothing architectures, we argue that the large numbers of processing cores and the high memory bandwidth available on GPUs have made them competitive in solving big data problems up to a certain scale. We further argue that, the techniques we have developed in this study for single computing node can be used as the building blocks to be integrated with existing MapReduce/Hadoop systems for larger scale problems to scale out. This is left for our future work. We would like to note that, while our research is motivated by practical needs on managing and processing big taxi trip data, many techniques can be applied to other types of spatial and relational data.

III. A FRAMEWORK FOR TRIP PURPOSE ANALYSIS USING SPATIAL QUERY PROCESSING

Typically each taxi trip is associated with a pair of pickup location/time and drop-off location/time as well as fare, distance and duration information. While the trip records do not tell their trip purposes directly, when associating the pick-up/drop-off locations and times with urban infrastructure data, such as street networks, Land Use Types (LUTs) and Points-of-Interests (POIs), the trip-purposes can be speculated. Although the identified trip purposes for individual trips may not be completely accurate, given the large number of taxi trips, the identified trip purposes are useful from a probability distribution perspective. We also believe identifying trip purposes from large-scale taxi trips is orthogonal and complementary to existing survey based (diary and/or GPS) trip purpose identification approaches where the accuracy is higher at the individual trip level but the numbers of trips are limited. The proposed approach represents a radical change from traditional labor-intensive transportation data collection to potentially deeper and more accurate understanding of urban dynamics with lower costs through ubiquitous sensing and computing intelligence.

Through a partnership with the NYC Taxi and Limousine Commission (TLC), we have access to the raw transaction data of NYC medallion taxicabs for eighteen months (2008-2010) that amounts to roughly 300 million GPS-based trip records. We also have access to the NYC MapPluto Tax Lot from NYC Department of City Planning (DCP) [12]. The MapPluto tax lot dataset contains rich land use information where each tax lot (polygon) is associated with a LUT. Currently there are 11 LUTs (see Table 4 in Section V for the list). A trip starts near a lot of *Family Buildings* (types 01 or 02) and ends near a lot of *Commercial/Office Buildings* (04) is likely to be work related. Similarly, a trip starts near a lot of *Transportation& Utility* (07) and ends near a lot of *Open Space & Outdoor Recreation* (09) is likely related to visitors outside of NYC. While it takes more domain knowledge and requires fine-tuning the combinations of the N*N (N=11) Origin-Destination types to identify more meaningful and interpretable trip categories, the most computationally expensive step is to associate each pickup and drop-off location with its nearest neighbor polygon. After the LUTs of the polygons at the pickup and drop-off locations of a taxi trip are derived through their nearest polygons, the trip can be aggregated to the corresponding statistics or histograms (one of N*N) for further analysis. As N is fixed and is typically small, the computing cost for this final aggregation step is just a fraction of a second even on a single CPU core and can be parallelized using the techniques presented in [1]. As such, in this study, we will focus on the spatial query step, i.e., searching for nearest polygons based on point-to-polygon boundary distances.

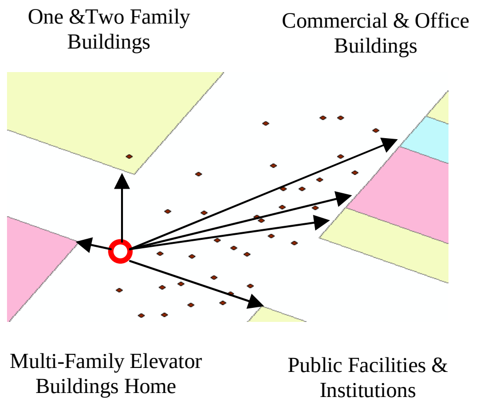
One &Two Family Buildings
Commercial & Office Buildings
Multi-Family Elevator Buildings Home
Public Facilities & Institutions

Fig. 2 Using LUT Label of Nearest Polygon (Tax Lot) of a Taxi Trip Record for Trip Analysis

As illustrated in Fig. 2, the polygons that represent the tax lots are spatially non-overlapping and are constrained by the city street network. As most taxi pickup and drop-off locations are along street segments, they are outside of tax lots. The exceptions might be due to GPS errors or arranged pickups/drop-offs in special cases. As such, it is natural to use the distance between a pickup/drop-off location (point) and the boundary a polygon (tax lot) as a measurement of likelihood that the trip is related to the LUT of the tax lot. While it might be more accurate by taking all the tax lots within a distance R into consideration, as a first step, we currently take only the nearest tax lot within a distance R into consideration. Extending 1NN to KNN is relatively straightforward from a computing perspective, which is left for our future work.

The observation has led us to develop a framework for trip purpose analysis using big taxi trip data, as shown in Fig. 3. The shaded components represent those that have been realized. Please note that incorporating temporal aggregation and filtering (the middle part of Fig. 3) is similar to what we have proposed in [1] from a technical perspective. The techniques can be used to analyze trip in specific time periods or during special events (e.g., new years, sport events). In our future work, we plan to incorporate Point-of-Interests (POIs) to further improve the accuracy of our trip purpose analysis. For tax lots in non-residential areas, there can be many POIs located in the same building in a tax lot and the POIs may have complex but interesting semantic relations that can potentially be useful for better accuracy. We also plan to use survey data in a machine-learning framework to improve the accuracy of our trip analysis. In all cases, the computing performance of the basic building block of the proposed framework in Fig. 3, i.e., searching the nearest polygons for all taxi trips, is the key in the success of trip purpose analysis using big taxi trip data. We next present the GPU-based spatial query processing techniques in the next section.

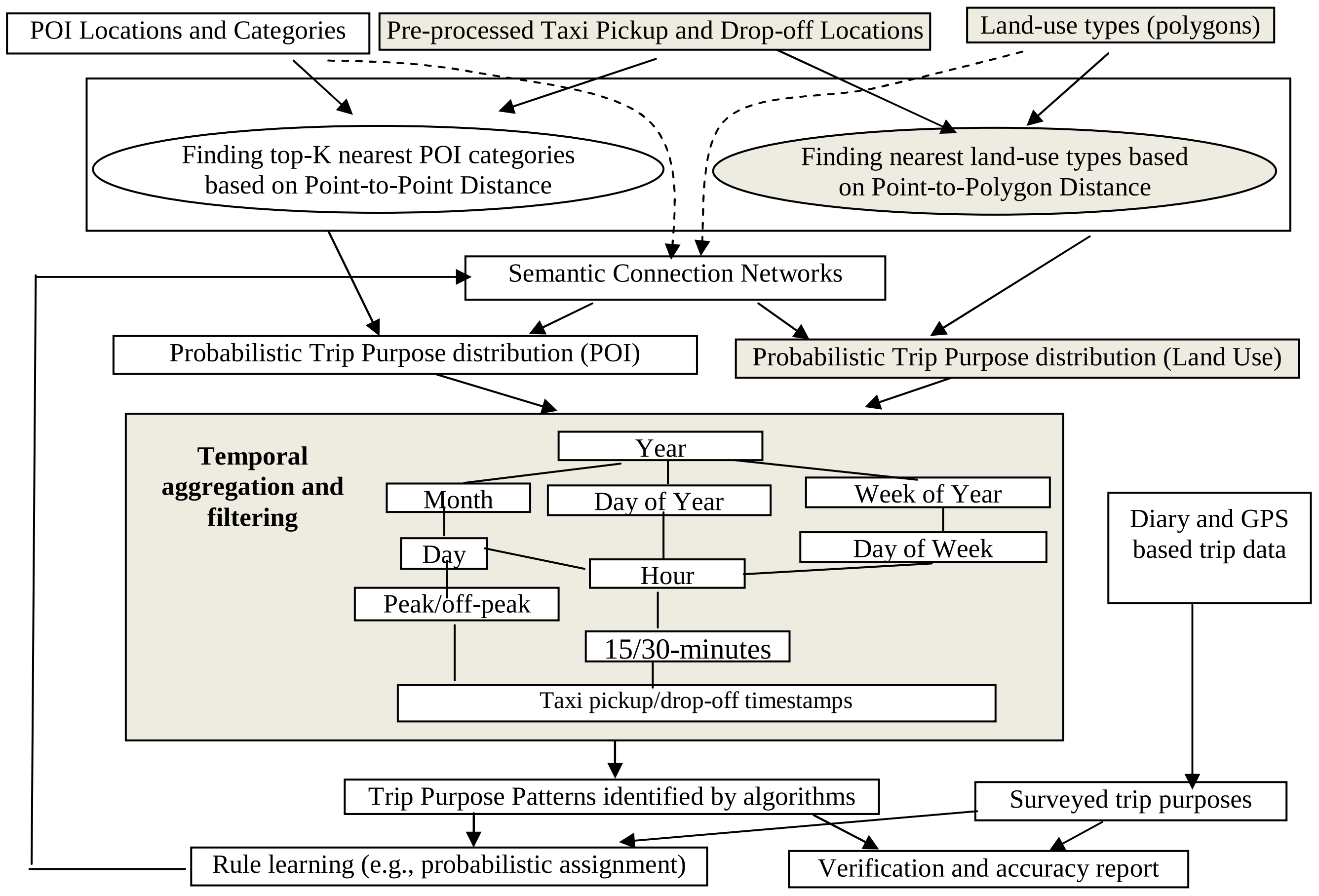
POI Locations and Categories
Pre-processed Taxi Pickup and Drop-off Locations
Land-use types (polygons)
Finding top-K nearest POI categories based on Point-to-Point Distance
Finding nearest land-use types based on Point-to-Polygon Distance
Semantic Connection Networks
Probabilistic Trip Purpose distribution (POI)
Probabilistic Trip Purpose distribution (Land Use)
Temporal aggregation and filtering
Year
Month
Day of Year
Week of Year
Day
Day of Week
Hour
Peak/off-peak
15/30-minutes
Taxi pickup/drop-off timestamps
Diary and GPS based trip data
Trip Purpose Patterns identified by algorithms
Surveyed trip purposes
Rule learning (e.g., probabilistic assignment)
Verification and accuracy report

Fig. 3 Framework of Big Taxi Trip Data Analysis

IV. PARALLEL DEISGNS AND IMPLEMENTATIONS ON GPGPUS

The high-level designs are illustrated in Fig. 4 which follows a spatial join framework [5] closely by using a grid-file structure for spatial indexing [1]. We next present the designs for the four modules in the spatial query processing, i.e., point indexing (to align points to grid cells), polygon indexing (to align expanded polygon Minimum Bounding Boxes –MBRs - to grid cells), spatial filtering (to pair up points with nearby polygons based on common grid cells) and spatial refinement (to associate each point with its nearest polygon based on point-to-polygon distances). As shown in the top part of Fig. 4, we store point coordinates and polygon vertex coordinates as arrays for better performance (e.g., being cache friendly on CPUs and coalesced memory accesses on GPUs). As detailed in [1], the boundaries of polygons and their rings are also stored as index arrays (i.e., PLI at the top-right part of Fig. 4).

For point indexing, based on the experiments reported in [1], as it is simpler and more efficient to index points using a flat grid-file structure than the multi-level quad-tree structure that we previous developed in [13], we have used the flat grid-file structure in this study. While we refer to the details provided in [1], which also uses the same point dataset for a different application, basically points are sorted by using row-major ordered cell-identifiers as keys and points with a same cell identifier are grouped into a cell. As such, a point index array (PTI in the top-left part of Fig. 4) is also used to store the starting positions of points in all the cells, in a way similar to the role of PLI.

For polygon indexing, the R-expanded MBRs of polygons, i.e., MBRs expanded by distance R along both directions, are also rasterized based on the same grid tessellation. It is clear that if a grid cell is not part of the expanded MBR of a polygon, then any of the point in the cell is at least R distance away from the polygon boundary and such cell-polygon pair should be excluded from subsequent spatial refinement. The GPU implementations of the first three modules can reuse the techniques presented in [1] as the primitives-based parallel designs and implementations are portable across different parallel hardware platforms. The details are omitted here due to space limit.

As shown in the middle of Fig. 4, for a cell-polygon pair (*C*,*P*) that should be sent for further spatial refinement based on true geometrical distances between points and polygons, the coordinates of points that fall within the cell *C* and the coordinates of polygon vertices can be retrieved from their respective coordinate arrays. As the shortest distance between a point and a polygon is defined as the smallest distance between the point and all the polygon edges, we can further reuse the shortest point-to-polyline distance computation module developed in [1] for this purpose. We do need, however, handle the neighboring vertices that belong to two different rings in a polygon in this particular application. As shown in the bottom of Fig. 4, we assign a (*C*,*P*) pair to a GPU computing block. Each thread is assigned to process a point which loops through all the polygon vertices to compute the shortest distance to the polygon. If a cell is paired with multiple polygons, then the polygon with the shortest smallest distance will be chosen to be associated with the point. The polygon identifier and the shortest distance will be assigned to each point. Although currently we have not used the computed shortest distances to adjust LUT probabilities for better accuracy, we plan to do so in our future work. As such, we have not used an obvious optimization of simply assigning the polygon identifier to all the points in a cell if only one polygon is paired up with the cell.

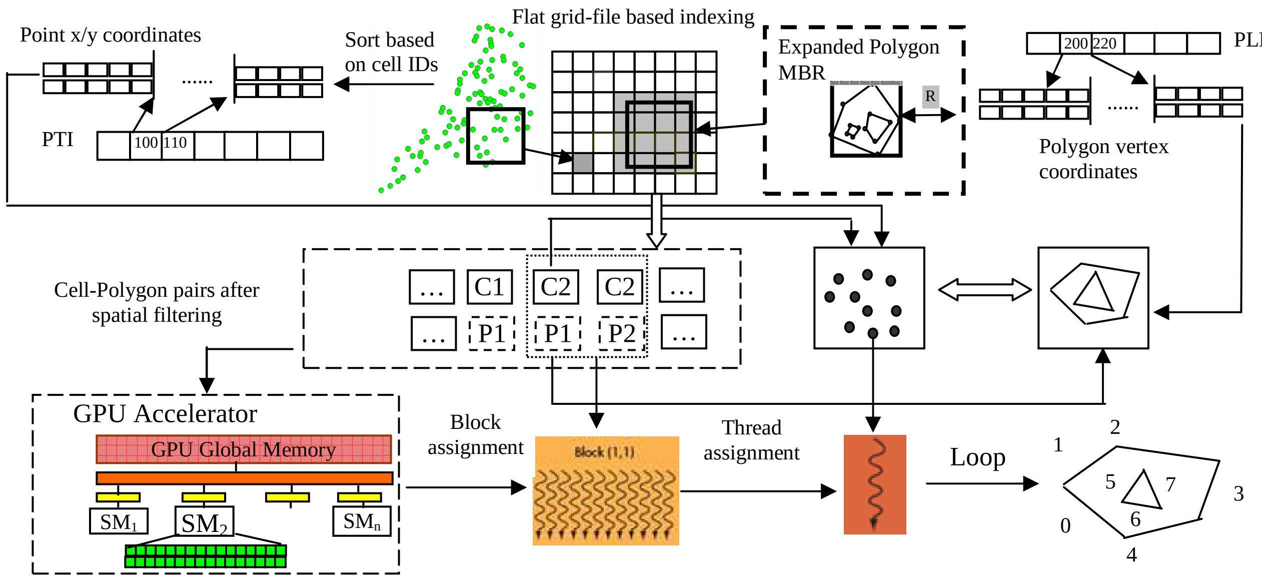

Fig. 4 High-Level Designs of GPU-based Parallel Spatial Query Processing to Associate Polygons with Points

Clearly, as *R* increases, the expanded polygon MBRs will likely to be more overlapped and a cell is likely to be paired up with more polygons. As such, the computing intensity increases as R goes up. We further note that, even for a large R value, it is possible that a grid cell is not paired up with any polygons. As such, our spatial query is not a nearest neighbor query in a strict sense, which requires find a nearest neighbor no matter how far way it is. Instead, the nearest neighbor polygon of a point in our approach is selected from polygons whose expanded MBR intersects with the cell that the point falls within, i.e., the cell is no more than *R* distance away from the MBR of the polygon. Please note that the rule does not guarantee that the shortest distance between a point and its nearest polygon is less than *R*. Although we can iteratively increase *R* until all points find their nearest polygon regardless *R* values to meet the conventional definition of nearest neighbor, we choose to use fixed *R* values in our experiments as we consider nearest neighbors are only meaningful within a certain distance buffer (as represented by *R*) in this particular application.

V EXPERIMENTS, RESULTS AND DISCUSSIONS

A Experiment Setup

All experiments are performed on a Do-It-Yourself (DIY) workstation equipped with a single Intel dual-core Core i5-650 CPU running at 3.2 GHZ, 8 GB GDDR3 memory and 500GB hard drive. Since the hardware support hyper-threading, the CPU appears to have four processing cores which are all used in our parallel implementations on CPUs. The CPU has 32KB L1 cache (per core) and 256KB L2 cache (per core) but there is no L3 cache for the CPU. The memory bandwidth is 21 GB/s. The total cost of all the parts used to assemble the workstation is around $1000 which put it in the lower end. The absence of L3 cache and the low memory bandwidth has significantly limited the machine's computing power when compared with high-end workstations. While the low-cost workstation is fairly weak in terms of computing power, we have quipped with an Nvidia GTX Titan GPU that has 2,688 cores (running at 877 MHZ), 6GB device memory and 288 GB/s memory bandwidth. We have compiled both the CPU and GPU source code with –O2 optimization flag for fair comparisons. We also mention that the total cost of the workstation (~$2000) is comparable or even lower than many computing nodes in cloud computing facilities which makes it possible to compare monetary cost.

Our experiments focus on two aspects, i.e., the runtimes of spatial query processing and the interesting patterns that can be derived from taxi trip data. For the first set of experiments, we will report the runtimes of the four modules using three different *R* values using both CPUs and GPUs in Section V.B. For the second set of experiments, we will report the numbers of taxi trips in each of the N*N combinations as an O-D matrix and provide some preliminary analysis on some of the potentially interesting patterns based on the resulting matrix in Section V.C.

B Results on Spatial Query Proceesing

Since the runtimes of experiments on pickup locations and drop-off locations are largely the same, we will report runtimes for pickup locations unless stated otherwise. It is clear that point indexing is independent of *R* values while the rest three modules are sensitive to different *R* values.

Table 1 Runtime Comparisons of Point Indexing

	CPU	GPU
Step 1: data loading from disk (ms)	26285.40	
Step 2: data preparation (ms)	1183.24	
Step 3: computing cell identifiers (ms)	221.43	385.17
Step 4: sort based cell identifiers (ms)	8177.81	588.31
Step 5: computing cell index array(ms)	1840.50	46.13

Table 1 lists the runtimes of the five steps in point indexing in milliseconds. Note that the first two steps are performed on CPUs. The GPU implementation of the point indexing module differs in the last three steps. Note that we load both pickup and drop-off locations from disks in Step 1 as they are stored in a same physical file. Step 2 checks data validity and performs some basic transformations. Although this step is easily parallelizable, since the runtime of this step is only a small fraction of the end-to-end runtime, we run this step in CPU sequentially for convenience. The reason that the GPU implementation has a higher runtime than CPU in Step 3 is that, point data is transferred from CPU to GPU in this step in the GPU-based implementation. Assuming that the CPU to GPU data transfer rate is 4 GB/s, transferring 170 million *8 bytes = 1.36GB data already takes 340 ms, which clearly dominates the GPU time in this step. From Table 1 we can see that, for the rest two steps, the GPU implementation is significant faster than the CPU implementation, i.e., 14X for Step 4 and 40X for Step 5.

Table 2 lists the runtimes of the rest of the rest of the three modules on both CPUs and GPUs in milliseconds under three *R* values, i.e., 50, 100 and 200 feet, respectively. Clearly, as expected, the runtimes increase as *R* values become larger. Table 2 also show that, he spatial refinement module that computes the distances between points and polygon boundaries is the most computing intensive one among all the four modules (including point indexing). The GPU implementations have achieved 16X-75X speedups among these three modules. For the spatial refinement module, the speedups vary from 24X-30X. The results in this module are more consistent than the other modules.

Table 2 Runtime and Speedup Comparisons among Other Three Modules Using Three R values

R (ft)		Polygon-indexing	Spatial Filtering	Spatial Refinement
50	CPU (ms)	2579	2175	613031
	GPU (ms)	161	46	25507
	Speedup	16.03X	47.20X	24.03X
100	CPU (ms)	4743	3921	789149
	GPU (ms)	293	74	29999
	Speedup	16.19X	49.96X	26.31X
200	CPU (ms)	46287	12287	1260588
	GPU (ms)	634	164	43067
		73.01X	74.90X	29.27X

To better understand the overall performance of the CPU-based implementations and the GPU-based implementations, we have listed the end-to-end runtimes under two scenarios, i.e., with and without including point data disk I/Os and the corresponding speedups. While we have parallelized all the important steps in the four modules, disk I/Os remain to be a bottleneck that is difficult to tackle in big data applications. The speedups listed in the right-most column of Table 3 represent the upper bounds that GPU computing can expect to achieve, after removing the disk I/O bottleneck (such as pre-loading or using flash drives). By including point data disk I/O times, as shown in the third column of Table 3, the realized speedups of the end-to-end runtimes in this study under the three R values range from 12X to 19X, which are still impressive. While we cannot include the runtimes of single thread implementations due to space limit, our results on the spatial refinement module (computing point-to-polygon distance and searching for nearest polygon) indicate that using all the 4-threads in the 2 CPU cores is about 1/3 better than single-thread (using a single core), i.e., 1.5X speedup. This may indicate that, limited cache capacity and low memory bandwidth on this low-end workstation may among the factors that prevent linear scalability with respect to processor cores and hardware threads, given that our data parallel designs have demonstrated better scalability on server grade CPUs [1].

Table 3 End-to-End Runtime and Speedup Comparisons With and Without Including Point Data Disk I/O time

R (ft)		With Point data disk I/O time	Without Point data disk I/O time
50	CPU (ms)	655493	629208
	GPU (ms)	54203	27917
	Speedup	12.09X	22.54X
100	CPU (ms)	835521	809236
	GPU (ms)	58859	32574
	Speedup	14.20X	24.84X
200	CPU (ms)	1356870	1330585
	GPU (ms)	72355	46070
	Speedup	18.75X	28.88X

We have not included direct comparisons with Hadoop-based implementations in this study as we are not aware of existing Hadoop-based implementations that have similar functionality. However, our early work presented in a technical report at [14] have included a serial CPU implementation using two popular open source geospatial software packages, i.e., libspatialindex [15] for R-Tree based polygon indexing and GDAL [16], for point-to-polygon distance computation. While the serial CPU implementation simply query the nearest polygon for each point iteratively, which leaves room for algorithmic improvements, the performance can be used as a baseline for an idealized comparison. Assuming that runtime of the serial implementation is Ts, then the best expected runtime for a Hadoop system with N computing nodes would be Ts/N, by excluding the overheads of network communication costs and disk I/Os for intermediate results. According to [14], for R=100, the end-to-end runtime for the serial CPU implementation to associate points with their nearest polygons is 110,093seconds. For verification purposes, the serial CPU implementation code has been made available online at [17]. In contrast, by adding up the runtimes of the four modules in the R=100 experiments, the end-to-end runtimes for our GPU implementation is Tg=58.856 and seconds including point data disk I/O time and Tg=32.574 seconds without including I/O time. In order for a Hadoop-based implementation to match the GPU-based implementation by adapting the serial implementation, even under the idealized assumption, the number of processing units would be N=Ts/Tg. Although N might be different when plugging in runtimes under different scenarios and accounting the differences among the CPUs, generally N should be in the order of 500-3000 based on the simple calculation. The computed N value is well above the numbers of computing nodes that are utilized by typical applications (in the order of dozes). The results may indicate that, GPUGPU computing can be attractive to practical big data applications with respect to end-to-end performance and monetary cost. On the other hand, the development cycle is much shorter for the serial implementation using open source packages, which can be more important in certain applications. Plugging the serial implementation into a Hadoop system by chunking points into segments is also relatively straightforward, which can be advantageous. The work reported in this study can serve as a case study to understand tradeoffs among different technologies in big data applications.

C Results on Taxi Trip Data Analysis

The output of our spatial query processing is a N*N matrix with each element in the matrix n[i][j] represents the number of trips from LUT i to LUT j. We have added 00 to indicate that LUT cannot be identified for either pickup location or drop-off location or both. As discussed earlier, as i=1..11 and j=1..11, these 121 combinations can be categorized into a few types of trip purposes for more domain-specific analysis; this is left for our future work. In this section, we will provide a preliminary analysis on the spatial query results which are listed in Table 4.

From the totals listed in the last column of Table 4, it is clear that the top-3 LUTs for pickup locations are 05 (*Commercial & Office Buildings*, 46.1 million), 09 (*Open Space & Outdoor Recreation*, 37.3 million) and 04 (*Mixed Residential & Commercial Buildings*, 28.5 million). Interestingly, based on the last row of Table 4, the top-3 drop-off LUTs are also 05 (44.6 million), 09 (38.9 million) and 04 (24.9 million), in the same order. These three LUTs cover about 2/3 of trips with respect to both pickup locations and drop-off locations and each of them has at least 20 million trips for both pickup and drop-off locations. LUT 03 (*Multi-Family Elevator Buildings*), 07 (*Transportation & Utility*) and 08 (*Public Facilities & Institutions*) are among the next tier LUTs with respect to the numbers of trips for both pickup and drop-off locations and they are in the range of 10-20 million. Trips that are covered by all the rest five LUTs are far below 10 million. The clear three-tier pattern makes it interesting for further studies.

Table 4 also suggests that, among the 13.1 million trips that start at LUT 07 (*Transportation & Utility*), the destination of the majority of the trips are LUT 05 (*Commercial & Office Buildings*, 4.1 million) followed by LUT 09 (*Open Space & Outdoor Recreation*, 2.5 million), LUT 07 itself (2.2 million) and LUT 04 (*Mixed Residential & Commercial Buildings*, 1.2 million). This may indicate passengers who arrive at NYC through public transportation (air, rail or bus) are mostly for business, leisure, transfer or coming back home (in this order). Some similar patterns can be derived from Table 4 for further analysis. Although subsequent validations are required for these patterns, the spatial query results are quite useful to stimulate hypothesis. While the results reported here are aggregated at the highest spatial (citywide) and temporal scale (yearly), our techniques allow incorporating spatial and temporal filtering for finer scale aggregations. We are also in the process of integrating visualization modules to visualize individual as well as aggregated query results for better interpretation and validate potentially interesting patterns to facilitate decision making.

Table 4 City-Level Origin-Destination Matrix of Numbers of Aggregated Trips in 2009 in NYC (in millions)

F/T	00	01	02	03	04	05	06	07	08	09	10	11	Total
00	0.151	0.078	0.123	0.197	0.297	0.622	0.072	0.184	0.181	0.231	0.044	0.027	2.209
01	0.012	0.054	0.081	0.133	0.168	0.237	0.035	0.054	0.085	0.164	0.011	0.022	1.057
02	0.089	0.108	0.253	0.417	0.795	0.831	0.122	0.153	0.316	0.525	0.035	0.063	3.707
03	0.260	0.263	0.715	2.035	3.004	4.593	0.543	0.979	1.192	2.769	0.249	0.279	16.880
04	0.579	0.411	1.112	3.373	5.905	7.465	0.984	1.309	2.210	4.336	0.424	0.441	28.549
05	0.711	0.586	1.333	4.844	6.496	19.486	1.445	3.813	2.964	3.083	0.864	0.477	46.103
06	0.121	0.057	0.161	0.526	0.764	1.197	0.204	0.213	0.309	0.245	0.063	0.063	3.922
07	0.073	0.365	0.344	0.913	1.235	4.099	0.261	2.246	0.724	2.528	0.152	0.182	13.122
08	0.160	0.198	0.493	1.350	2.083	2.444	0.285	0.470	0.947	1.423	0.128	0.104	10.086
09	0.279	0.442	0.968	2.512	3.472	2.313	0.274	1.665	1.576	23.121	0.267	0.440	37.330
10	0.074	0.038	0.090	0.193	0.431	0.812	0.070	0.138	0.157	0.129	0.042	0.031	2.206
11	0.024	0.029	0.068	0.205	0.314	0.501	0.048	0.140	0.099	0.387	0.015	0.069	1.899
Total	2.533	2.630	5.743	16.697	24.965	44.599	4.343	11.365	10.760	38.940	2.295	2.198	167.068

LUT labels (defined in Section III): 00-unknown, 01- One & Two Family Buildings, 02 - Multi-Family Walk-Up Buildings, 03- Multi-Family Elevator Buildings, 04- Mixed Residential & Commercial Buildings, 05- Commercial & Office Buildings, 06-Industrial & Manufacturing, 07-Transportation & Utility, 08-Public Facilities & Institutions, 09 - Open Space & Outdoor Recreation, 10 - Parking Facilities, 11-Vacant Land.

VI. Conclusions

In this study, we aim at utilizing the massively data parallel processing power provided by modern GPUs to speed up spatial query processing on large-scale taxi trip data for aggregated trip purpose analysis. By integrating the parallel designs and implementations of GPU-based spatial indexing and query processing techniques, we have successfully developed a set of techniques to compute the nearest polygon of both pickup and drop-off locations in a taxi trip record and aggregate taxi trips based on land use types of their nearest polygons. Experiments have shown that our GPU implementations can complete such complex spatial queries in about 50-75 seconds using an inexpensive commodity GPU device. The performance is 10X-20X higher than the host machine with an Intel dual-core CPU when all the cores and hardware supported threads are fully used. Preliminary analysis on the resulting trip count matrix has demonstrated interesting patterns and opens the doors for future work to validate the patterns and discovery new patterns through more complex spatial, temporal and spatiotemporal queries.

References

[1] J. Zhang, S.You and L. Gruenwald (to appear). Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs. Information Systems (Elsevier) , in-press. Mancuscript online at http://www-cs.ccny.cuny.edu/~jzhang/papers/IS14_DOLAP_Manuscript.pdf

[2] S. Shekhar, V. Gunturi et al (2012). Spatial big-data challenges intersecting mobility and cloud computing. In Proc. ACM MobiDE'12.

[3] A.Cary, Z. Sun, V. Hristidis, et al. (2009). Experiences on Processing Spatial Data with MapReduce. In Proc.SSDBM'09, 302-319.

[4] S. Zhang, J. Han, Z. Liu, et al. (2009). SJMR: Parallelizing spatial join with MapReduce on clusters. In Proc. IEEE CLUSTER'09 Workshops.

[5] E. H. Jacox and H. Samet (2007). Spatial join techniques. ACM Transaction on Database System 32(1), Article #7

[6] J. Wolf, R. Guensler and W. Bachman. (2001). Elimination of Travel Diary: Experiment to Derive Trip Purpose from Global Positioning System Travel Data. TRB #1768, 125-134

[7] P. R. Stopher, and S. P. Greaves (2007). Household travel surveys: Where are we going? Transportation Research Part A: Policy and Practice 41(5): 367-381

[8] Metropolitan Transportation Authority. Subway and Bus Ridership. http://www.mta.info/nyct/facts/ridership/index.htm

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

[10] I.H. Lee, Y. J. Lee, H. Choi, et al. (2012). Parallel data processing with MapReduce: a survey. SIGMOD Record 40(4):11-20

[11] B. Kirk and W.-m. W. Hwu (2012). Programming Massively Parallel Processors: A Hands-on Approach (2nd ed.). Morgan Kaufmann.

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

[13] J. Zhang, S.You (2012). Speeding up large-scale Point-in-Polygon test based spatial join on GPUs. In Proc. ACM BigSpatial'12,23-32.

[14] J. Zhang, S. You and L. Gruenwald (2012). Speeding High-Performance Spatial Join Processing on GPGPUs with Applications to Large-Scale Taxi Trip Data. Technical Report. Online at http://www-cs.ccny.cuny.edu/~jzhang/papers/nnsp_tr.pdf

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

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

[17] http://www-cs.ccny.cuny.edu/~jzhang/spp2p.htm