Accepted by Information Systems (Elsevier) http://dx.doi.org/10.1016/j.is.2014.01.005

Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs

Jianting Zhang, Department of Computer Science, the City College of New York, New York, NY, 10031, USA, jzhang@cs.ccny.cuny.edu
Simin You, Department of Computer Science, CUNY Graduate Center, New York, NY, 10016, USA, syou@gc.cuny.edu
Le Gruenwald, School of Computer Science, the University of Oklahoma, Norman, OK 73071, USA, ggruenwald@ou.edu

Abstract

With the increasing availability of locating and navigation technologies on portable wireless devices, huge amounts of location data are being captured at ever growing rates. Spatial and temporal aggregations in an Online Analytical Processing (OLAP) setting for the large-scale ubiquitous urban sensing data play an important role in understanding urban dynamics and facilitating decision making. Unfortunately, existing spatial, temporal and spatiotemporal OLAP techniques are mostly based on traditional computing frameworks, i.e., disk-resident systems on uniprocessors based on serial algorithms, which makes them incapable of handling large-scale data on parallel hardware architectures that have already been equipped with commodity computers. In this study, we report our designs, implementations and experiments on developing a data management platform and a set of parallel techniques to support high-performance online spatial and temporal aggregations on multi-core CPUs and many-core Graphics Processing Units (GPUs). Our experiment results show that we are able to spatially associate nearly 170 million taxi pickup location points with their nearest street segments among 147,011 candidates in about 5-25 seconds on both an Nvidia Quadro 6000 GPU device and dual Intel Xeon E5405 quad-core CPUs when their Vector Processing Units (VPUs) are utilized for computing intensive tasks. After spatially associating points with road segments, spatial, temporal and spatiotemporal aggregations are reduced to relational aggregations and can be processed in the order of a fraction of a second on both GPUs and multi-core CPUs. In addition to demonstrating the feasibility of building a high-performance OLAP system for processing large-scale taxi trip data for real-time, interactive data explorations, our work also opens the paths to achieving even higher OLAP query efficiency for large-scale applications through integrating domain-specific data management platforms, novel parallel data structures and algorithm designs, and hardware architecture friendly implementations.

Keywords: OLAP, Parallel Design, GPU, Multi-core CPU, Spatiotemporal Aggregation, Spatial Indexing, Spatial Join, Large-Scale Data

1. INTRODUCTION

With the increasing availability of locating and navigation technologies on portable wireless devices, huge amounts of location data are being captured at ever growing rates. For example, the approximately 13,000 taxicabs in the New York City (NYC) equipped with GPS devices generate more than half a million taxi trip records per day. Cell phone call logs represent a category of data at an even larger scale [1][2]. Also as visitors travel around the world more frequently, location-dependent social networks such as Foursquare [3], and location-enhanced social media such as text posted to Wiki sites [4], and images and videos posted to Flickr and YouTube [5], can also potentially generate large-amounts of spatial and temporal data. All the three types of data have a few features in common: (1) they are produced and collected by commodity sensing devices and are rich in data volumes in urban areas; and (2) they are a special type of spatial and temporal data with an origin location and a destination location in the geo-referenced space domain and a starting time and an ending time in the time domain. However, the intermediate locations between origins and destinations are either unavailable, inaccessible or unimportant. Compared with traditional geographical data collected by government agencies for urban planning and city administration purposes, these data can be more effective to help people understand the real dynamic of urban areas with respect to spatial/temporal resolutions and representativeness. We term such data as Ubiquitous Urban Sensing Origin-Destination data, or U^2SOD data, for notation convenience [6]. Despite the close relationships between U^2SOD data and Spatial Databases (SDB) [7] and Moving Object Databases (MOD) [8], our experiences have shown that traditional disk-resident and tuple/row oriented spatial databases and moving object databases are ineffective in processing large-scale U^2SOD data for practical applications including multidimensional aggregations, one of the most important modules in Online Analytical Processing (OLAP) [9].

Considerable work on developing efficient data structures and algorithms has been proposed for multidimensional aggregations on CPU uniprocessors in the past few decades [10]. Modern hardware architectures increasingly rely on parallel technologies to increase the processing power due to various limits in improving the speeds of uniprocessors [11]. Unfortunately, existing data structures and algorithms that are designed for serial implementations may not be able to effectively utilize the parallel processing power of modern hardware, including multi-core CPUs and many-core Graphics Processing Units (GPUs) [11]. Despite the fact that parallel hardware is already available in the majority of commodity computers, there is still relatively little work in exploiting such parallel processing power for OLAP queries, especially in the areas of spatial and temporal aggregations of large-scale geographical data where complex join operations are required in the aggregations. Examples of these are counting the number of taxi pickups at each of the community districts or census blocks (spatial aggregation), generating hourly histogram of drop-offs near the JFK airport (temporal aggregation) and computing numbers of trips between Time Square and Central Park in morning peak hours (OD aggregation).

In this study, we report our work on developing a data management framework for large-scale U^2SOD data and a set of data parallel designs of spatial and temporal aggregations that can be realized on both multi-core CPUs and many-core GPUs in an OLAP setting. Our experiments on aggregating the approximately 170 million taxi trip records in NYC in 2009 have demonstrated the effectiveness of the proposed framework and parallel techniques. By utilizing parallel spatial joins [12] to support efficient online processing, we are able to achieve real-time responses for spatial, temporal and spatiotemporal aggregations at different hierarchical levels.

Compared with traditional approaches that rely on relational databases and spatial databases for aggregations, our techniques have reduced the OLAP query response times from hours to seconds. This makes it possible for urban geographers and transportation researchers to explore the large-scale origin-destination data in general and taxi trip data in particular in an interactive manner.

The rest of the paper is arranged as follows. Section 2 introduces the background, motivation and related work. Section 3 presents a parallelization-friendly data management framework for managing large-scale origin-destination data with a focus on taxi trip records. Section 4 provides the details on the parallel designs and implementations of spatial and temporal aggregations on both multi-core CPUs and many-core GPUs. Section 5 reports the experiment results. Finally Section 6 is the conclusion and future work.

2. BACKGROUND, MOTIVATION AND RELATED WORK

The increasingly available location data generated by consumer wireless portable devices, such as GPS, GPS enhanced cameras and GPS/WiFi/Cellular enhanced mobile phones, has significantly changed the ways of collecting, analyzing, disseminating and utilizing urban sensing data. Traditionally city government agencies are responsible for collecting various types of geographical data for city management purposes, such as urban planning and traffic control. The data collection is usually done through sampling, typically at coarse-resolutions, and questionnaire-based investigations which often incur long turn-around times. In contrast, as consumer mobile devices become ubiquitous, similar data obtained from GPS-traces and mobile phone call logs has much finer resolutions. With the help of privacy and security related technologies, the aggregated records from such ubiquitous urban sensing data can be enormously helpful in understanding and addressing a variety of urban related issues. Research groups from both academia (e.g., MIT Sensible City Lab) and industries (e.g., IBM Smart Planet Initiative and Microsoft Research Asia) have developed techniques to utilize such data and understand the interactions among people and their locations/mobility at the city level (e.g., Beijing [13], Boston [14] and Rome [1]), social group level (e.g., friends [15] and taxi-passenger pairs [16]) and individuals level [17]. However, most of the existing studies focus on the data mining aspects of such ubiquitous urban sensing data through case studies while largely leaving the data management aspects untouched. Lacking proper data management techniques can result in significant technical hurdles in making full use of such data to address outstanding societal concerns. In this study, we focus on efficiently aggregating large-scale taxi trip records to better understand human mobility and facilitate transportation planning by developing high-performance spatial, temporal and spatiotemporal aggregation techniques in an OLAP setting.

OLAP technologies are attractive to explore the possible patterns from large-scale taxi trip records and other types of origin-destination data. As the taxi trip data has spatial dimensions and temporal dimensions for both pickup and drop-off locations and conventional dimensions (e.g., fare and tip), taxi trips can be naturally modeled as spatial, temporal and spatiotemporal data which requires synergizing existing research on Spatial OLAP [18][19] and Temporal OLAP [18][20] or their combinations [18]. Due to the popularity of geo-reference data, there are increasing research and application interests in Spatial OLAP. However, most of them focus on data modeling and query languages [18][21][22][23], and applications on top of spatial databases and Geographical Information Systems (GIS) [24][25][26][27]. A few sophisticated indexing and query processing algorithms to speed up certain analytical operations, such as consolidation/aggregation, drill-down, slicing and dicing, have been proposed [10][28][29][30][31]. Spatial OLAP applications on top of spatial databases and GIS, while easy

to implement, impose additional I/O and computational overheads which may further slow down spatial and temporal aggregations and may not be suitable for applications that involve a large number of data records like our taxi trip application. We also note that the existing research on Spatial OLAP mostly targeted at the traditional computing framework, i.e., disk-resident systems on uniprocessors based on serial algorithms, which makes it incapable of handling large-scale data on parallel hardware architectures.

Our experiments using the open source PostgreSQL database have shown that the performance of spatial aggregations on a large-scale dataset, which contains hundreds of millions of taxi trip records, using the traditional disk-resident database systems, is too poor to be useful for our applications. We note that spatial queries are supported in PostgreSQL through the PostGIS extension [32]. The Appendix at the end of the paper lists 16 SQL statements (Q1-Q16) that are involved in a database based implementation of spatial and temporal queries, where tables *t* and *n* represent the taxi trip records data and street network data, respectively. Note that queries Q1 through Q8 are used for spatial associations (including indexing and spatial join). Q9 and Q10 are used for indexing materialized spatial relationships, i.e., PUSeg and DOSeg are indexed as relational attributes. Q11 and Q12 are used for spatial aggregations based on the materialized spatial relationships. Finally, Q13 and Q14 are used for temporal indexing and Q15 and Q16 are used for temporal aggregations. On a high-end computing node running PostgreSQL 9.2.3, Q5 took dozens of hours. We note that Q5 is already an optimized SQL statement by using the non-standard "SELECT DISTINCT ON" clause in PostgreSQL and approximating the nearest-neighbor query using the ST_DWithin function and the "ORDER BY *distance*" clause. Obviously the performance is far from satisfactory for online OLAP queries. While we are aware of the fact that certain optimization techniques, such as setting proper parameters and data partitioning, can potentially improve the overall performance, we believe that Spatial OLAP queries based on traditional database systems cannot achieve the performance level that we are aiming at for the data at the scale using existing technologies. Our additional experiment results have also revealed that the performance can be drastically improved by utilizing large main-memory capacities and GPU parallel processing [33][34]. This has motivated us to investigate techniques in boosting the performance of spatial, temporal and spatiotemporal aggregations by making full use of modern hardware that has already been equipped with commodity personal computers.

We refer the readers to [9] for a brief review on parallel OLAP computation. We note that existing work on parallel OLAP mostly focused on parallelization on shared-nothing architectures while leaving parallelization on shared-memory Symmetric Multiprocessing (SMP) architectures, including both multi-core CPUs and many-core GPUs, largely untouched. The number of processing cores on both single-node CPUs and GPUs is fast increasing. The mainstream Intel CPUs and Nvidia GPUs have 4-8 and 512 cores, respectively. Devices based on the Intel Many Integrated Core (MIC) architecture (such as Xeon Phi 5110p) have 60 cores [35] and devices based on Nvidia Kepler architecture equipped with close to 3,000 cores [36] are currently available on the market. These inexpensive devices based on shared-memory SMP architectures are cost-effective and relatively easy to program. We believe it is an attractive alternative to cluster computing in solving many practical large-scale data management problems when compared to MapReduce based cloud computing where computing resources are often utilized inefficiently [37]. Despite the fact that shared-nothing based architectures are often considered having better scalability than shared-memory based ones, we argue that, from a practical perspective, higher scalability can be achieved by integrating the two architectures

when necessary. Fully utilizing the parallel processing power of SMP processors (including both CPUs and GPUs) will naturally improve the overall system performance in a cluster computing environment using grid or cloud computing resources. As a first step, we currently focus on parallel aggregations on multi-core CPUs and many-core GPUs equipped in a single computing node, i.e., in a personal computing environment that is more suitable for interaction-intensive applications such as OLAP queries.

There are a few pioneering works on using multi-core CPUs and many-core GPUs for OLAP queries including aggregations. The design and implementation of the HYRISE system [38] have motivated our work in many aspects, such as column-oriented physical data layout, data compression and in-memory data structures. However, most of the existing systems including HYRISE are designed for traditional business data and do not support geo-referenced data. There are also several attempts in using GPUs for OLAP applications with demonstrable performance speedups [10][39][40]. However, again, they do not explicitly support spatial or spatiotemporal aggregations which are arguably more computationally intensive. Furthermore, while previous studies have shown that parallel scan based GPU implementations can be effective in processing data records in the order of a few millions , the number of data records in our application is almost two orders of magnitude larger which makes GPU implementation more technically challenging.

Our designs and implementations of spatial and temporal aggregations heavily rely on parallel primitives that are supported by several parallel libraries on both CPUs and GPUs. Parallel primitives, such as *sort*, *scan* and *reduce* [41][42], refer to a collection of fundamental algorithms that can be run on parallel machines. The behaviors of popular parallel primitives are well-understood. In particular, we have used the open source Thrust library [36] that comes with Nvidia CUDA SDK [43] on GPUs and the open source Intel TBB [44] package from Intel on CPUs extensively. It is beyond the scope of this paper to provide a comprehensive review of the parallel primitives that we have utilized in this study and we refer the interested readers to [42] for more details. A brief introduction to several parallel primitives that are involved in our GPU implementations of the aggregations is provided online [45]. The parallel sorting primitive that we have used in this study comes from the GNU libstdc++ Parallel Mode library which was derived from the Multi-Core Standard Template Library (MCSTL) project [46]. In addition, our multi-core CPU implementation of spatial associations utilizes the Vector Processing Units (VPUs) on CPUs to boost its performance. While several pioneering studies have tried to exploit the Single Instruction Multiple Data (SIMD) parallel processing power on VPUs by calling lower level hardware-specific APIs (SIMD intrinsics) [47][48][49], we take advantage of the Intel ISPC open source compiler [50] that has recently become available. The ISPC compiler supports programming SIMD units in a way similar to CUDA based GPU programming which is much more productive. To the best of our knowledge, we are not aware of any previous work on utilizing VPU's SIMD processing power for spatial associations by speeding up geometrical computation.

3 U^{2}SOD-DB: MANAGING ORIGIN-DESTINATION (OD) DATA IN DBMS

Almost all taxi cabs in cities of developed countries have been equipped with GPS devices and different types of trip related information are recorded. For example, more than 13,000 GPS-equipped medallion taxicabs in the New York City (NYC) generate nearly half a million taxi trips per day and approximately 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 (Metropolitan Transportation Authority) ridership statistics [51]. Taxi trips play important roles in everyday lives of residents and visitors of NYC as well as any major city worldwide. The raw service data has a few dozens of attributes such as pick-up/drop-off location and time as well as fare/tip/toll amounts. In this study, we generalize the taxi trip data as a special type of OD data and present the design and implementation of the U^{2}SOD-DB system that is designed for managing U^{2}SOD data on modern hardware.

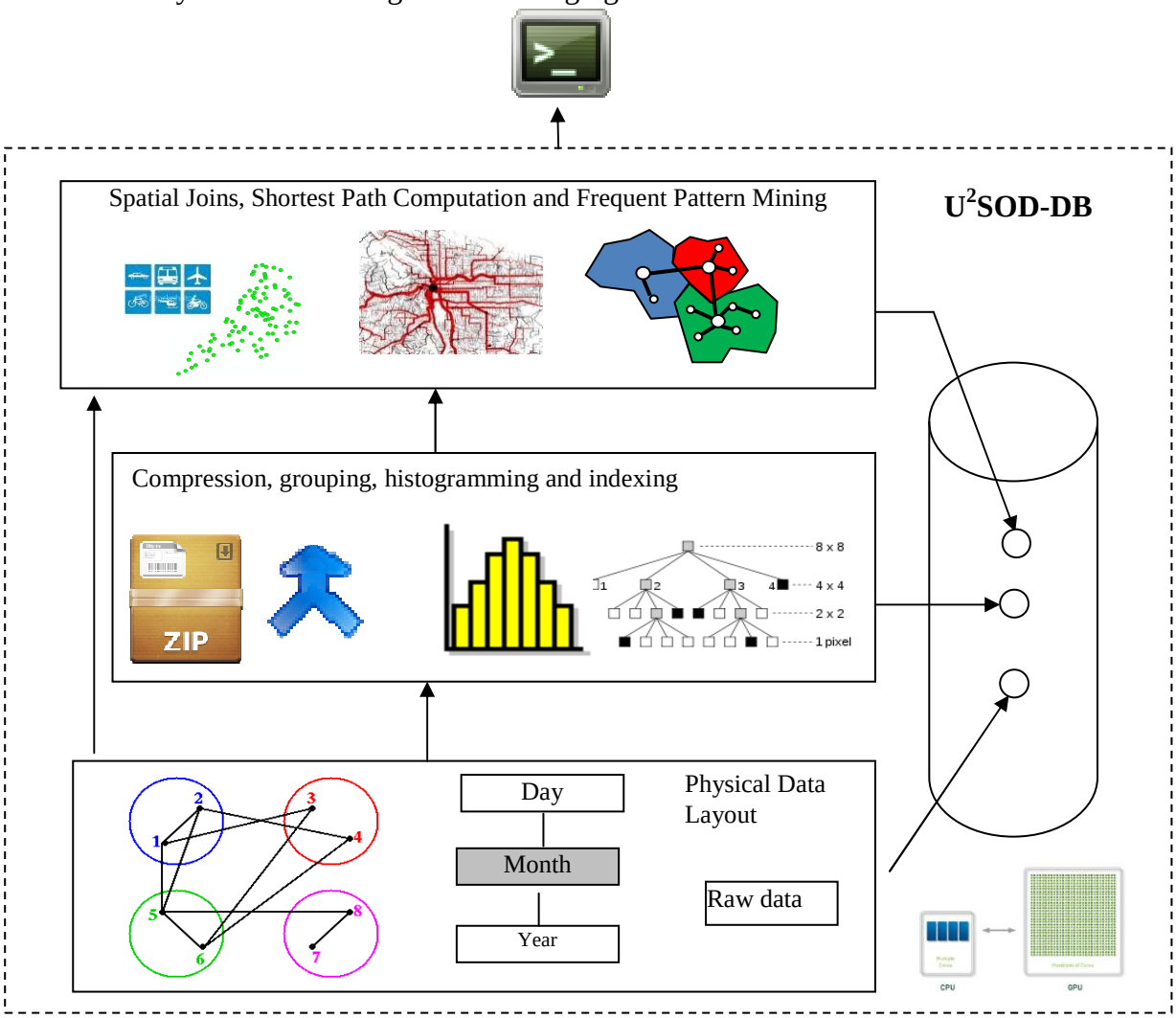

Fig. 1. Illustration of U^{2}SOD-DB Prototype System Components and Interactions

Our design of U^{2}SOD-DB has three tiers. The lowest tier is closely related to physical data layout and we have adopted a time-segmented, column-oriented data layout approach. The raw data are first transformed into binary representations and attributes are clustered into groups based on application semantics. The data corresponding to the attribute groups at a certain time granularity are stored as a single database file with the relevant metadata registered with the database system. We assume one or more database files can be streamed into CPU main memory as a whole to maximize disk I/O utilization. Multi-core CPU processors can access the data files in parallel once they are loaded into the CPU main memory. They can also be transferred to GPU

global memories should the system determine that GPU parallel processing is more advantageous. The middle tier is designed to support efficient data accesses and provides some commonly used routines, such as compression, histogramming and indexing. The top tier is more application specific. For spatiotemporal aggregations in an OLAP setting, the current U^{2}SOD-DB supports efficient spatial joins between OD data and urban infrastructure data, such as road networks and administrative regions. An illustration of U^{2}SOD-DB prototype system components and interactions is provided in Fig. 1 [6]. For the remaining of this section, we provide technical details on the data layout and timestamp compression and a high-level overview of spatial and temporal aggregations before their parallel designs and implementations details are presented in Section 4.

3.1 Time-Segmented Column-Oriented Data Layout

As shown in Fig. 2 [6], we categorize the attributes that are associated with trip records into several groups and data of the attributes in the same group are stored in a single database file by following the column-oriented layout design principle: attributes in the same group are likely to be used together and thus it is beneficial to load them into main memory as a whole to reduce I/O overheads. Given a fixed amount of main memory, as the attribute field lengths of individual groups are much smaller than the lengths of all attributes, more records that are related to analysis can be loaded into main memory for fast data accesses. In general, the column-oriented data layout design improves traditional tuple-based physical storage in relational databases by avoiding reading unneeded attribute values into main memory buffers and subsequently increasing the number of tuples that can be read into mmeory in a single I/O request. We also note that combining several attribute groups into one and extracting attributes from multiple groups to form materialized views can be beneficial for certain tasks. For example, in Fig. 2, attribute group 5 (start_x, start_y, end_x and end_y) can be considered as a materialized view of the attribute group 2 (start_lon, start_lat, end_lon and end_lat) by applying a local map projection to the latitude/longitude pairs. Since the projected data are frequently used in calculating geometric and shortest path distances and map projections are fairly expensive, materializing attribute group 5 can significantly improve system performance. Another example to demonstrate the utilization of materialized views on the physically grouped attributes is verifying the recorded trip times (indicated by trip_time in group 6) with computed trip times (by subtracting pickup time from drop-off time in group 3) by materializing the respective attributes in the two groups. We further note that among the attributes in the original dataset shown in Fig. 2, some of them can be derived from the others. For example, both start/end zip codes (group 9) and addresses of pickup and drop-off locations (group 8) can be derived from start/end latitudes and longitudes (group 2) through reverse geocoding or other related techniques.

After determining the data layout by grouping attributes (vertical partitioning), the next issue is to determine the appropriate number of records in database files so that these files can be efficiently streamed among hard drives, CPU main memory and GPU device memory. The sizes of the database files should be large enough to reduce system overheads as much as possible but small enough to accommodate multiple database files simultaneously in CPU/GPU memories so that typical operations can be completed in the designated memory buffers. At the same time, the numbers of records should correspond to certain time granularities as much as possible. In our design, we use month as the basic unit (temporal granularity) to segment taxi trip records (horizontal partition). Given that there are about half a million taxi trip records per day in NYC, assuming that an attribute group has a record length of 16 bytes (e.g. four attributes with each

represented by a 4-bytes integer), the database file would be 16*0.5*30=240 megabytes. Since the main memory capacity in our experiment system is 16 gigabytes, the database file size is appropriate although other sizes might be suitable as well.

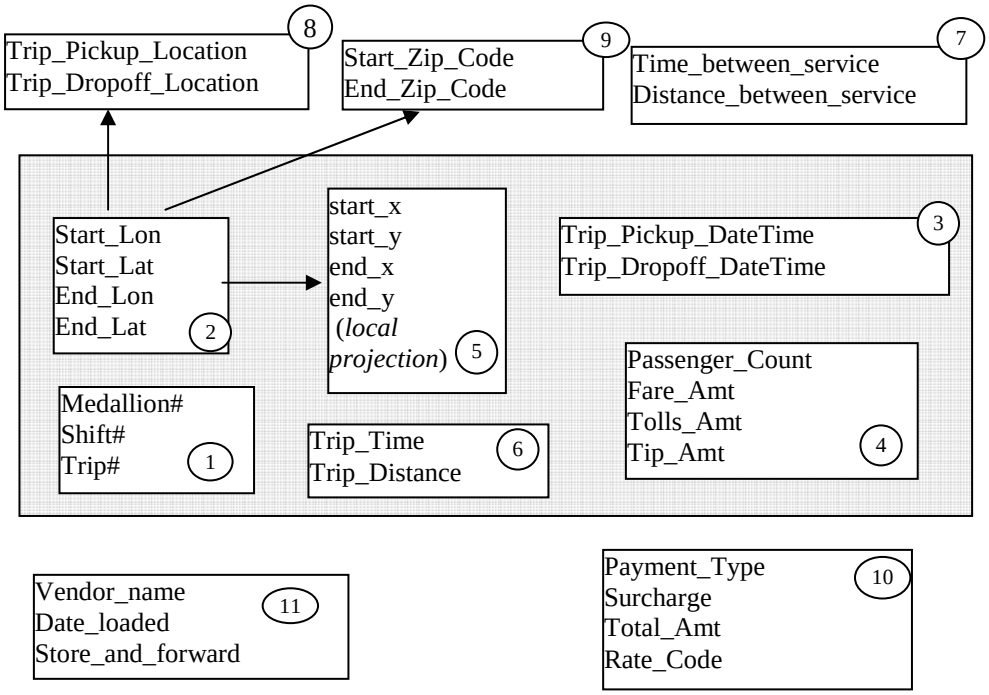

Fig. 2. Column-Oriented Physical Data Layout in U^{2}SOD-DB

3.2 Timestamp Compression and Temporal Aggregations

The text format of the pickup and drop-off times converted from relational databases like "2009-01-17 23:52:34" takes 20 bytes. While the format can be easily converted to "*struct tm*" in the standard C language to satisfy the needs of all temporal aggregations (e.g., month, day of week) on CPUs, we found that the data structure takes 56 bytes on 64-bit Linux and 44 bytes on 32-bit Linux platforms which may be too much from a memory footprint perspective. Furthermore, the time structure cannot be used on GPUs directly which brings a significant compatibility issue. Our solution is to compress the pickup and drop-off times (PUT and DOT) into 4-byte (32 bits) memory variables using the following bit layout starting from the most insignificant bit: 6 bits for second (0-59), 6 bits for minute (0-59), 5 bits for hour (0-23), 5 bits for day (0-30) and 4 bits for month (0-11). The remaining 6 bits (0-63) can be used to specify the year relative to a beginning year (e.g. 2000) which should be sufficient for a reasonably long study period. Retrieving any of the year, month, day, minute and second fields can be easily done by bitwise operations and integer operations which are efficient on both CPUs and GPUs. The straightforward technique has reduced memory footprint to 1/5 (4/20) and is friendly to both CPUs and GPUs.

At the first glance, the design does not support temporal aggregations based on "day of week" and "day of year" very well as these two fields are not explicitly stored in our design as in "*struct tm*" in the standard C library. Computing the values of these two fields, while feasible on CPUs (by using C/C++ *mktime* function), can incur significant overheads when the number of

records is huge. For example, our experiments have shown that computing "day of week" alone for 170 million records can take 22 seconds, i.e., 100+ CPU cycles per timestamp on average. However, we would like to draw attention to the fact that timestamps can be aggregated by "year+month+day" before they are further aggregated according to "day of week" or "day of year". Since the possible combinations of (year, month, day) in a reasonably long period are limited (in the orders of a few thousands), they can be aggregated to "day of week" or "day of year" in a fraction of a millisecond using the C/C++ *mktime* function on CPUs. The design also eliminates the need for GPU implementation to compute "day of week" or "day of year" from "year+month+day" which is nontrivial.

3.3 Overview of Spatial and Temporal Aggregations

While many operations and analytical tasks can be performed on U^{2}SOD data, spatial and temporal aggregations are among the fundamental ones. In a way similar to efficient OLAP operations for decision support on relational data, high-performance spatial and temporal aggregations are crucial in effectively supporting more complex analytical operations on OD data. Fig. 3 illustrates the general framework and example spatial and temporal aggregations that are supported by our U^{2}SOD-DB design. Most of the temporal aggregations (e.g. daily and hourly) and some of the spatial aggregations (e.g. grid based) can use data-independent schemes, while more complex aggregations rely on the schemas provided by infrastructure data such as road network and administrative hierarchies. Our designs and implementations to be presented in Section 4 focus on complex spatial aggregations. We note that, as shown in Fig. 3, once the OD locations (e.g. pickup and drop locations in the taxi trip data) are associated with street segments or different types of zones, spatial, temporal and spatiotemporal aggregations can be reduced to simple relational aggregations without involving expensive spatial and/or temporal operations any more. The U^{2}SOD-DB architecture is designed to support multiple types of spatial aggregations by providing a common spatial indexing and spatial filtering framework to efficiently pair subsets of relevant datasets before applying specific refinement approaches for associating individual data items, e.g., based on Nearest Neighbor (NN) search or Point-In-Polygon (PIP) test. In this study, we focus on spatially associating taxi locations with segments in road networks by locating the nearest street segment within distance R for each point location. After the 170 million NYC tax trip locations are associated with the LION road network dataset published by the NYC Department of City Planning (DCP) [52], as street segments are nicely associated with quite a few types of polygon zones as shown in Fig. 3, they can be used for further relational aggregations.

It is beyond the scope of our study to implement all the types of spatial, temporal and spatiotemporal aggregations that have been modeled in the literature [53][18]. Instead, we focus on the aggregations along the spatial and temporal hierarchies shown in Fig. 3. We divide an aggregation into two phases: the spatial and temporal association phase and the relational aggregation phase. The association phase, typically implemented as a join, can be performed either offline or online. The advantage of materializing spatial, temporal or spatiotemporal relationships offline is that, as computing the relationships typically is expensive, directly accessing the materialized relationships can significantly improve the overall performance. However, when dynamic query criteria are imposed (such as those based on taxi fare and tip), offline materialization becomes infeasible and fast real-time online aggregations become critical. In addition, when spatial aggregations are combined with temporal aggregations (i.e., spatiotemporal aggregations) at arbitrary levels, the possible number of aggregations grows

quickly which makes offline materialization less attractive due to disk storage, I/O and maintenance overheads. Online aggregations are more desirable in such cases.

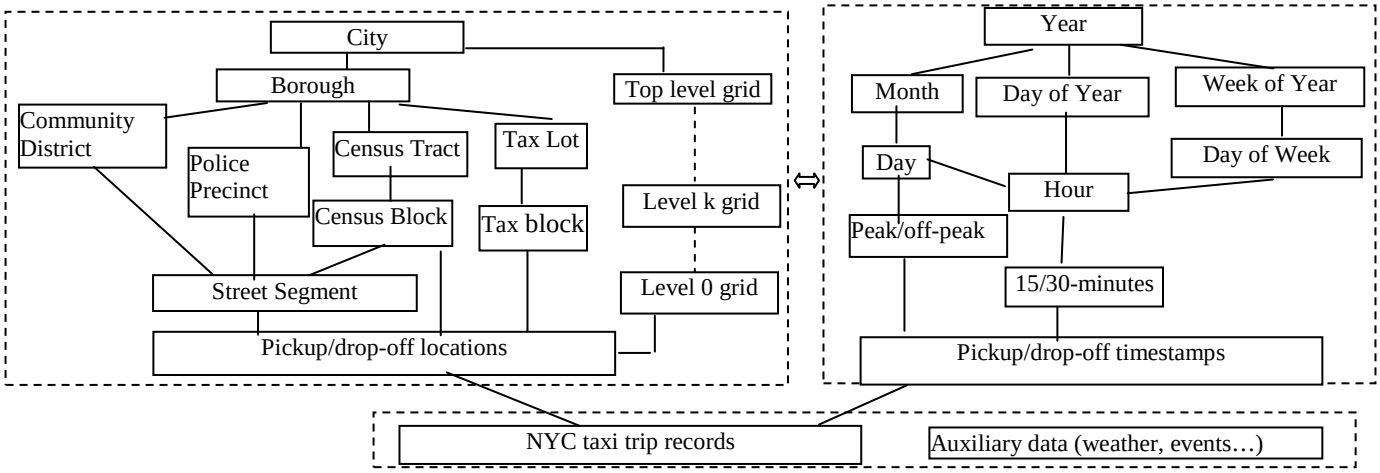

Fig. 3. Example Spatial and Temporal Aggregations in U^{2}SOD-DB

4 PARALLEL DESIGNS AND IMPLEMENTATIONS OF SPATIAL ASSOCIATION

The U^{2}SOD-DB architecture supports both serial and parallel designs and implementations of the spatial and temporal aggregation operations discussed above. In this study, we propose a set of data parallel designs that can be implemented on both multi-core CPUs and many-core GPUs. For spatial associations, in addition to providing the design and implementation details of the Multi Level Quadrants (MLQ) based approach presented in [34] that focuses on many-core GPUs, we have also provided an alternative set of designs and implementations of a Flatly Structured Grid (FSG) based approach that focuses more on multi-core CPUs. While these designs and implementations will be detailed in subsections 4.1 through 4.5, Table 1 provides an overview of the modules and their available implementations on multiple parallel computing platforms. Note that we use "/" to indicate that the implementation is applicable to both designs while we use "+" to indicate similar but different implementations are required for the MLQ and FSG designs. The rationales are provided when we describe the details of each module. Among the four modules in both approaches, the Point Indexing and Polyline Indexing modules are used to partition points into groups so that only spatially close points and polygons are paired up in the Spatial Filtering module before each point is associated with its nearest polyline in the Spatial Refinement module. Provided that a good spatial filtering strategy is available, it is possible to reduce the nearest neighbor computing overhead from O(N*M) to O(N) where N is the number of points and M is the number of polylines, respectively.

The primary reason that motivates us to develop the FSG based design and implementation for point indexing is to overcome the memory limitation on GPUs (currently limited to 6GB) for larger point datasets. The implementation is based on the parallel sorting algorithm provided by the GNU libstdc++ Parallel Mode library [46] which is very efficient on multi-core CPUs. For the Polyline Indexing module and Spatial Filtering module, we only provide a parallel primitive based design (whose implementation is based on Thrust) because their runtimes are relatively insignificant on large datasets when compared to the other two modules. By changing the underlying computing platform from CUDA to TBB, the parallel primitive based implementations based on Thrust can be compiled to CUDA code on GPUs and

compiled to TBB code on multi-core CPUs. The downside of this highly portable solution is that, in a way similar to using Standard Template Library (STL) [54], there are library overheads which can be significant in certain cases, especially for using TBB on multi-core CPUs where the Thrust library has not been extensively optimized for multi-core CPUs. However, for the Spatial Refinement module, we provide both a native CUDA implementation on GPUs and a native TBB implementation on multi-core CPUs as we want to reduce the overheads due to parallel libraries and minimize the overall response times. As detailed in Section 5, the modules can be organized into different configurations under different scenarios and the performance of the configurations can be further compared. We next provide the details of the designs and implementations of the four modules in the following subsections.

Table 1. List of Parallel Design and Implementation Choices

	Multi-core CPU Implementations	GPU Implementations
1 Point Indexing	GNU Parallel (FSG)	Thrust over CUDA (MLQ)
2 Polyline Indexing	Thrust over TBB (FSG)	Thrust over CUDA (FSG/MLQ)
3 Spatial Filtering	Thrust over TBB (FSG)	Thrust over CUDA (FSG+MLQ)
4 Spatial Refinement	TBB(with ISPC) (FSG)	CUDA (FSG/MLQ)

4.1 Point Indexing Using Multi-Level Quadrants

As illustrated in Fig. 4, the strategy of the MLQ based point indexing is to partition the point data space in a top-down, level-wise manner and identify the quadrants with a maximum of *K* points at multiple levels. While the point quadrants are being identified level-by-level, the remaining points get more clustered, the numbers of remaining points become smaller, and the data space is reduced. The process completes when either the maximum level is reached or all points have been grouped into quadrants. In the example shown in Fig. 4, we first sort all points at level 1 using their Z-order [58] code and count the number of points under each level 1 quadrant. As quadrant 2 has only 11 points which is less than K=20, its key (2) and number of points under it (11) are identified and the points are excluded from further processing. We use a (key, #of points) pair to represent a quadrant in Fig. 4. The same procedure is applied to the remaining points at level 2 to identify two quadrants (4,9) and (7,9) and at level 3 to identify 6 quadrants (9,7), (10,9), (11,8), (12,5), (13,8), (14,7), in an iterative manner. After the numbers of points in all quadrants are derived, the starting positions of the first points in the quadrants can be computed in parallel easily by using prefix-sum [41][42] for accesses to the points later. An advantage of limiting the maximum number of points in a quadrant to K is to facilitate load balancing in parallel computing. As it shall be clear after we introduce the spatial filtering and refinement modules, when the numbers of points and the numbers of polyline vertices are bounded, the workload of parallel processing elements is also bounded and no processing elements can dominate the whole process.

The design is highly data parallel and can be implemented using a few parallel primitives as detailed in [34] where the point indexing module is also used for point-in-polygon test based spatial joins. The design is implemented on top of the Thrust parallel library that provides all the necessary parallel primitives. Previous studies have shown that the parallel sorting primitives implemented in Thrust are capable of handling hundreds of millions of data items per second and are highly efficient [55]. Our experiments have shown that the most expensive step in this module is to sort points based on their quadrant identifiers before counting the numbers of points

in the quadrants and determining whether the points in the quadrants should be excluded for further partition. By making full use of high-performance GPU-based sorting parallel primitives, as shown in Section 5.2, the GPU-based implementation of the MLQ approach for point indexing has achieved 13X speedup over CPU-based serial implementation.

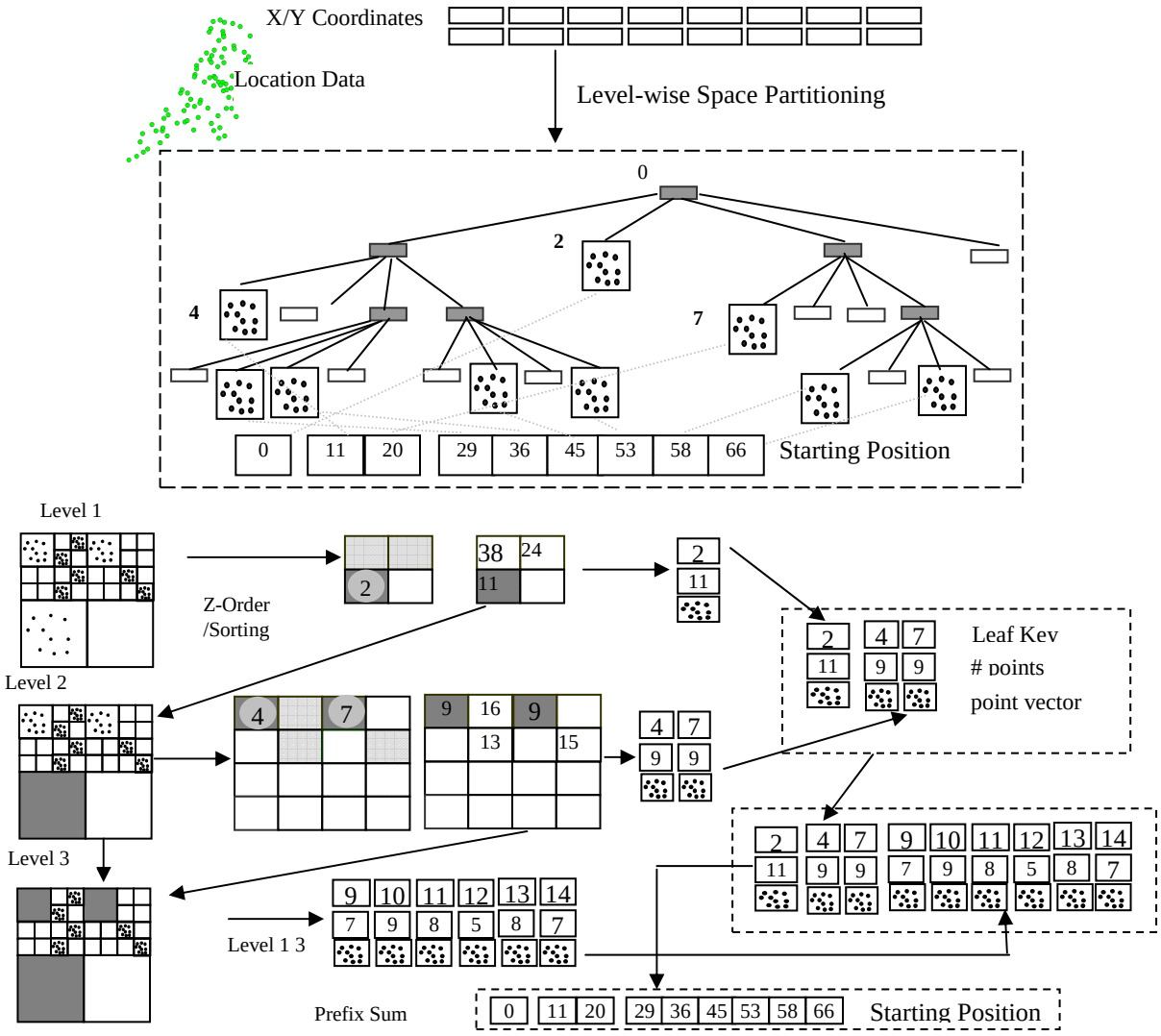

Fig. 4. Parallel-Primitive Based Point Indexing Using MLQ Design

However, while the underlying radix sort algorithm itself is scalable on shared memory systems (including both CPUs and GPUs), the applicability of the parallel sort primitives to large-scale point datasets is limited by GPU memory capacity (currently 6 GB). Our experiments have shown that the GPU implementation of the Point Indexing module based on the MLQ design allows a maximum number of points in the order of 150-200 million on an Nvidia Quadro 6000 device. While it is possible to use pinned memory on CPUs to "virtually" expand the GPU memory, our experiments suggest that extensive data communication (partially due to the nature of the underlying radix sort algorithm) incurs significant overheads which makes the technique impractical. In addition, our previous experiments have shown that copying all the point data

from CPU memory to GPU memory incurred 25% of overhead which is quite significant. While the GPU-based design and implementation is still useful for reasonably sized point datasets, it is necessary to seek alternative solutions to efficiently index large point datasets. While our CPU-based point indexing approach on multi-core CPUs that will be presented in the next subsection overcomes these problems, we note that several research efforts are available to support GPU-based sorting on CPU memory and external memory (e.g. [56]) and we plan to incorporate these techniques once they become available to make the MLQ based point indexing technique more scalable.

4.2 Point Indexing Using Flatly Structured Grids

We design an alternative approach for point indexing that completely runs on multi-core CPUs. As CPUs can have much larger memory capacity (up to hundreds of GBs on commodity workstations) than GPUs, the approach will eliminate the memory constraints to a certain extent. In addition, there is no need to copy data from CPU memory to GPU memory anymore which can potentially reduce end-to-end runtime considerably. Grid files are among the most classic techniques for spatial indexing [57]. The FSG design for point indexing on multi-core CPUs is based on a flat grid-file structure which is significantly simpler than the MLQ based design. Given a grid resolution, we can generate a key based on the coordinates of a point, i.e., deriving a Space Filling Curve (SFC) code based on row-major order or Z-order [58]. This step is highly parallelizable and is expected to be very fast as only element-wise operations, i.e., operations apply to all vector elements in parallel, are involved. The most important step in the FSG approach is parallel sorting of the point records based on keys. Our implementation is based on the GNU libstdc++ Parallel Mode library [46]. Based on our experiments, the implementation is memory efficient and is effective in making full use of multi-core CPU hardware. In addition to being applicable to large point datasets whose volumes are beyond the capacity of GPU memory, eliminating the need to transfer data back-and-forth between CPU and GPU memories can potentially improve end-to-end response time of the FSG approach on CPUs. As detailed in Section 5.3, the new approach is significantly more efficient than straightforward porting of the MLQ based GPU design and implementation to multi-core CPU, a feature offered by the Thrust library. The efficiency is largely due to the simplified design using a flatly structured grid. However, the simplified design also loses the load balancing feature provided by the MLQ approach, as the numbers of points in grid cells can potentially be unbounded. However, we argue that, when the grid resolutions are sufficiently fine (which is typically true in our applications using millions of grid cells), the chance of extreme cases where points in a few grid cells dominate the whole computation (in the Spatial Refinement module) is very small on multi-core CPUs. This is especially true on CPUs where the number of processing units (cores) is typically small (in the order of a few to a few dozens) and tasks can be dynamically balanced by, for example, using the scheduling modules provided by TBB explicitly or implicitly.

4.3 Polyline Indexing

Polylines can be indexed in a similar way as polygons based on their Minimum Bounding Boxes (MBBs). While many approaches have been proposed in the serial computing setting where data partition based approaches (such as R-Trees) are preferred [57], it remains unclear what is the best way to index polylines and polygons in parallel settings. In this study, we index polylines that represent street segments also based on grid files. As it shall be clear when the Spatial Filtering module is introduced in Section 4.4, after rasterizing the expanded polyline MBBs to grid cells, binary searches can be applied to pair point quadrants (MLQ approach) and

point cells (FSG approach) with polylines for refinement. As the number of polylines is typically much smaller than the number of points in spatial associations, we assume polyline MBBs can fit into GPU memory completely and thus we will only provide a GPU-based implementation that is independent of the MLQ and FSG designs.

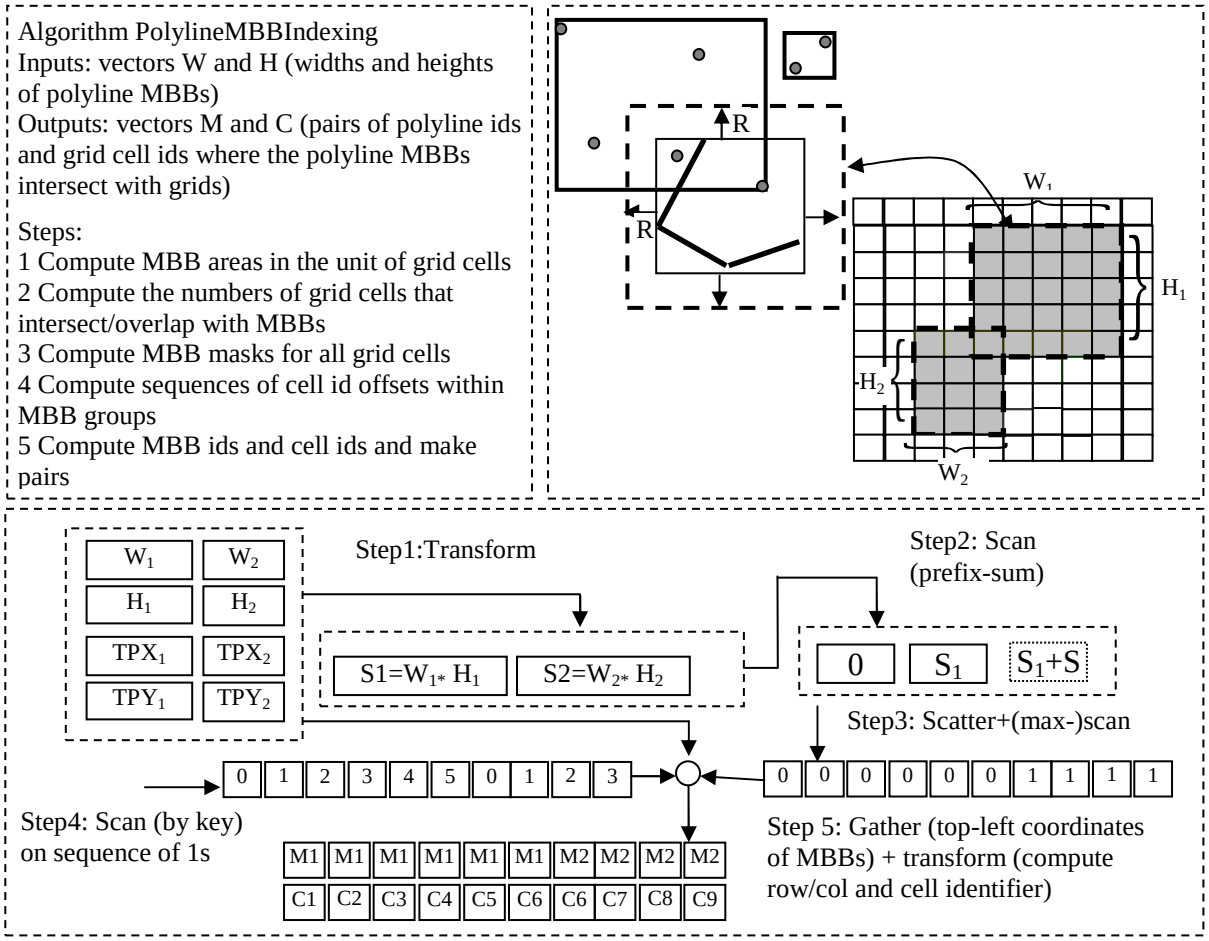

Fig. 5. Parallel-Primitive Based Polyline MBB Indexing

To support locating the nearest polyline for a point (x,y) within a query window width of *R*, as shown at the top-right part of Fig. 5, it is sufficient to examine all the polylines whose expanded MBBs intersect with the point. Assuming the MBB of a polyline is (x1,y1,x2,y2), the necessary condition for the polyline to be at most *R* distance away from the point is that the point intersects with the rectangle (x1-R,y1-R,x2+R,y2+R). Subsequently, for a group of points in a quadrant or a cell, the necessary condition for at least one of the points that are at most *R* distance away from a polyline is that the bounding box of the point group intersects with the expanded MBB of the polyline (x1-R,y1-R,x2+R,y2+R). The observation requires rasterizing the expanded MBBs of polylines, which we introduce next through an example with two expanded MBBs shown in the bottom part of Fig. 5. Assuming that the widths and heights of the two MBBs are W1, W2 and H1 and H2, and the coordinates of their top-left corners are (TPX1,TPY1) and (TPX2,TPY2), the first step is to compute the numbers of the cells that the two MBBs cover by using a *Transform* primitive. After computing the starting positions of the

first cell in the output cell vector through a *Scan (prefix-sum)* primitive in Step 2, the identifiers of the MBBs are scattered to the output cell vector and they are subsequently propagated along the vector using a *(max) Scan* primitive in Step 3. Using the sequences of cells in the MBBs generated in Step 4 and the MBB identifier vector derived in Step 3, Step 5 first retrieves the top-left coordinates of the MBBs and adds the local offsets of the cells they cover to compute the global cell identifiers for all cells in parallel by using a *Transform* primitive again. The algorithm is sketched in the top-left part of Fig. 5. We note that, for the derived multi-level point quadrants in the MLQ approach, the same procedure can be applied to rasterize the quadrants to grid cells.

4.4 Spatial Filtering

The Spatial Filtering module requires two similar but different implementations for the MLQ and FSG approaches. While the MLQ requires rasterizing the resulting quadrants as discussed previously and pair polylines and point quadrants indirectly through grid cell identifiers, the point grid cells can be directly paired with polylines in the FSQ approach, both through parallel binary searches as indicated by the dotted lines in the middle of Fig. 6A and Fig. 6B. In Fig. 6A (top), vectors VQQ and VQC keep the 1-to-many relationship between point quadrants and grid cells, and vectors VPP and VPC keep the 1-to-many relationship between polyline MBBs and grid cells. Identifiers of point quadrants in vector VQQ and identifiers of polyline MBBs can be paired through common grid cell identifiers in vectors VQC and VPC. In Fig. 6B (bottom), vector VGC used in the FSG approach is equivalent to VQC used in the MLQ approach and polyline MBBs are paired with grid cells directly. It is clear that the set of operations needed in the MLQ approach is a superset of those in the FSG approach. Although the implementation of the FSG approach for spatial filtering is simpler, unlike the MLQ approach that is designed explicitly for load balancing, it can potentially suffer from load unbalancing as the numbers of points in grid cells can vary significantly while the numbers of points in point quadrants are bounded by threshold K in the MLQ design. This may cause workload unbalancing in the refinement phase. However, similar to point indexing, we argue that when the grid resolutions are sufficiently high and the total number of cells is significantly larger than the number of processing units, load balancing can be achieved to a certain degree by dynamically scheduling (cell, MBB) pairs to parallel processing units.

4.5 Spatial Refinement

Recall that our goal is to compute the nearest polylines of points that are at most R distance away. After the spatial filtering phase, each point group (multi-level quadrant or grid cell) is associated with a set of polyline identifiers. What needs to be done in the Spatial Refinement module is to identify the polylines that are nearest to each of the points in the group by computing and ordering the distances between the point and the candidate polylines. Here the distance between a point and a polyline, as illustrated in Fig. 7, is canonically defined as the shortest distance between the point and all the line segments in a polyline. When the point is projected to a line segment, if the projection point falls between the two ends of the line segment, the point-to-line-segment distance is the distance between the point and the projection point (perpendicular distance); otherwise the point-to-line-segment distance will be the shorter of the distances between the point and the two ends of the line segment. Clearly, the spatial refinement module is more computation intensive than the rest of the modules as quite many floating point computations are required. For each point in a point group, looping through all the line segments of all the polylines whose expanded MBBs intersect with the bounding box of the point group is required in our design.

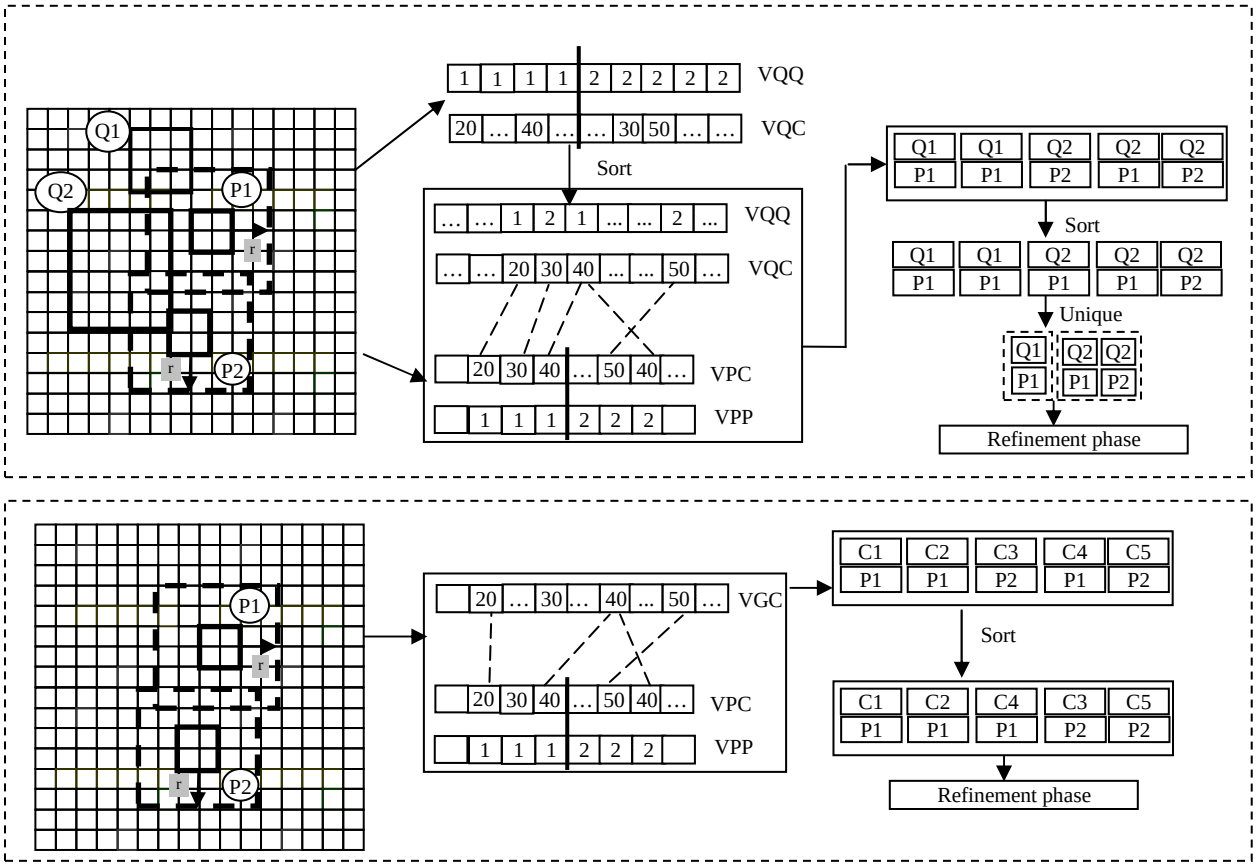

Fig. 6. Parallel Primitives based Spatial Filtering in MLQ (Top) and FSG (Bottom)

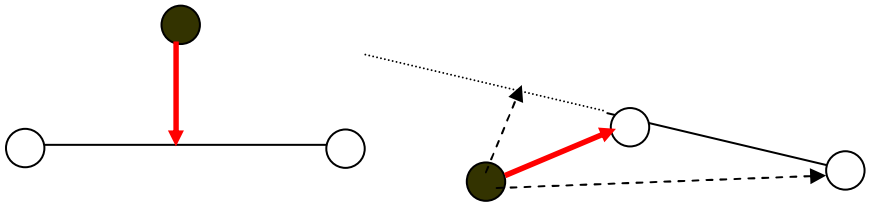

Fig. 7, Illustration of Computing Shortest Distance between a Point and a Line Segment

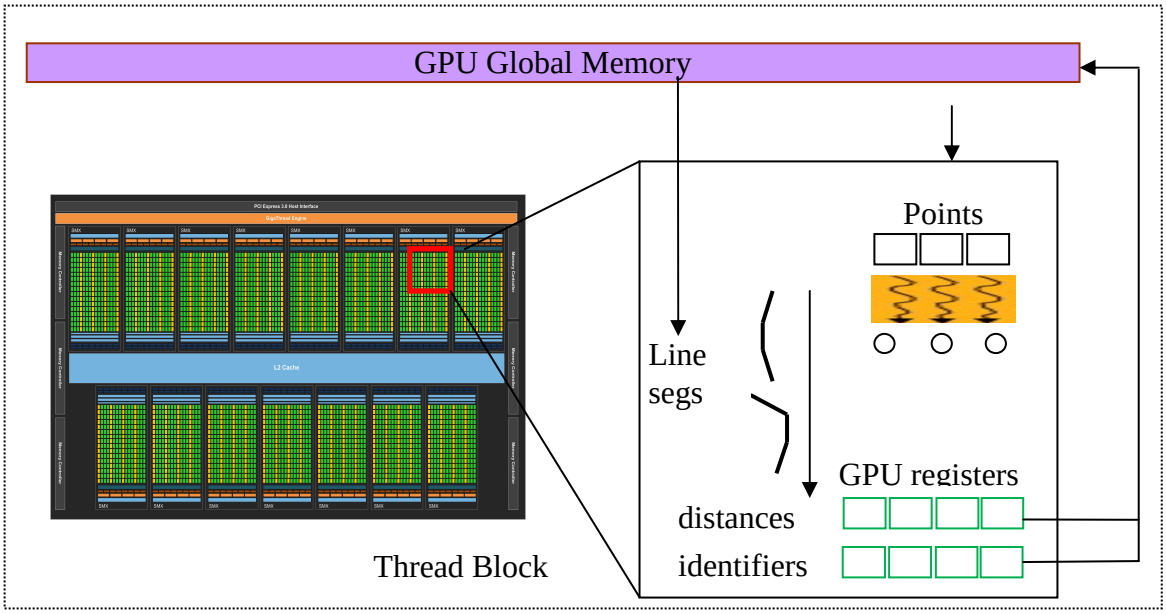

Fig. 8. Illustration of Neighbor-Polyline-Searching based Spatial Refinement on GPUs

Our GPU implementation of the refinement module uses CUDA directly to achieve better performance for two reasons. First, there are some unavoidable library overheads when using the Thrust parallel library. Second, more importantly, the refinement design involves multi-level loops and it does not fit the parallel primitives designed mostly for 1D vectors very well. In our GPU implementation, as shown in Fig. 8, we assign each point quadrant to a thread block and assign points in the quadrant to threads in the thread block. Threads within the computing block process points in the point quadrant in parallel and each thread loops through all the paired polylines and their line segments to compute the shortest distance. The identifier of the polyline with the shortest distance will be associated with the point. We note that the scheme is similar to the nested loop join in relational databases. Although nested loop joins are typically considered inefficient in disk-resident relational databases, as all GPU threads in a thread block access a continuous memory chunk that holds the point coordinates and all threads access the same line segment during a looping step on GPUs, the memory accesses are highly coalesced. Similarly, as the result of each point is a pair of polyline identifier and the shortest distance, the thread that processes the point knows exactly where to output the results. In a way similar to read accesses to point coordinates, neighboring threads also access neighboring vector elements of the outputs and memory accesses are also highly coalesced. Unlike CPUs, GPUs have very limited cache capacities and rely on coalesced memory accesses to hide high memory access latencies and achieve high performance. Our GPU implementation is thus able to achieve high performance due to the coalesced memory accesses in addition to utilizing GPU's excellent floating point computing power. We would like to note that while the GPU implementation is applicable to both the MLQ and FSG designs (as indicated in Table 1) as the inputs are in the same format for both designs, the implementation may incur different workloads and produce different results. This is because the pairs in the MLQ approach are at the quadrant level where quadrants can have different sizes although the numbers of points are bounded by K, while the pairs in the FSG approach are at the grid cell level where grid cells are no larger than quadrants and the numbers of points are unbounded. As such, there will be more points in the quadrants paired with polylines in the MLQ approach than the points in the grid cells paired with polylines in the FSG

approach. This is very similar to the well-known fact in spatial filtering that using coarse grids will incur more false positives, although the MLQ approach uses quadrants with variable sizes at different levels (but still larger than the grid cells in the FSG approach in this study). We expect the workload for the implementation based on the MLQ design will be heavier than that based on the FSG design which is verified in Section 5.3.

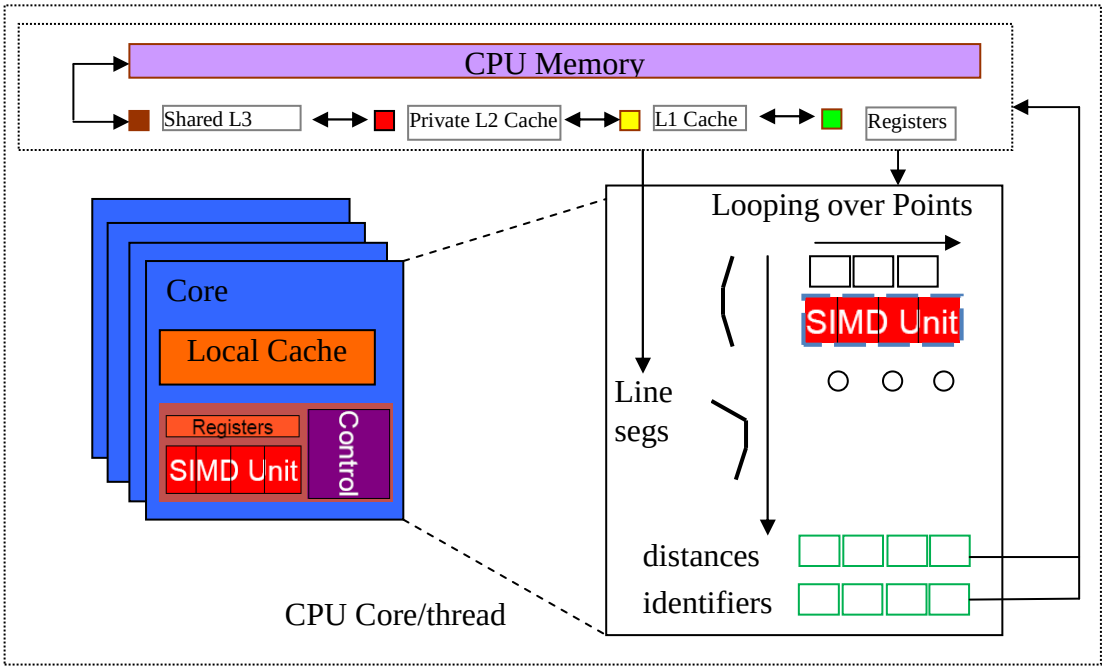

Fig. 9. Illustration of Neighbor-Polyline-Searching based Spatial Refinement on CPUs Equipped with Vector Processing Units (VPUs)

Our multi-core CPU implementation of the refinement module is based on the Intel open source TBB package [44] that typically performs better for computing tasks that are not well balanced. As shown in Fig. 9, we formulate processing a point cell as a task and let TBB handle the mapping between tasks and CPU threads. In runtime, TBB assigns a group of tasks to a CPU thread and lets the CPU thread loop through all the corresponding point cells. For each point cell, the thread loops through all the points in the cell and loops through all the line segments in the paired polylines. When compared with the GPU implementation, we can see that a CPU thread does much more work than a GPU thread. In addition, reading and writing to CPU memory are automatically cached by CPU caching subsystem which makes programming much easier. As an optimization of the multi-core CPU based implementation, we have developed a module to utilize the VPUs (Vector Processing Units) that are available on modern CPUs. SIMD based parallel processing on VPUs are considered closely related to the Single Instruction Multiple Thread (SIMT) model on CUDA enabled GPUs [36]. SIMD width has increased from 128-bit (4-way) in SSE (Steaming SIMD extensions) to 256-bit (8-way) in AVX (Advanced Vector Extensions) and both are available on mainstream CPUs. In addition, the gap between GPU SIMD width (currently 1024-bit, 32-way in a warp) and VPU SIMD width is rapidly decreasing (for example, Intel MIC VPUs have a 512-bit, 16-way SIMD width [35]). As such, exploiting VPUs for high performance on CPUs is becoming increasingly important. Our implementation utilizes the ISPC complier [50] that is available from Intel as an open source package. Unlike

CUDA threads that have their own program counters and thus allow complex control logic, all the SIMD elements of VPUs need to follow the same execution path which makes their utilization less flexible. Another difference is that, while CUDA allows much larger logic SIMD width (which is limited by the maximum number of threads in a computing block – currently in the order of thousands), the SIMD width need to match the VPU physical SIMD width of multi-core CPUs. As such, our design is to partition the points in a cell into chunks and use the VPUs to process the points in batches through explicit looping as shown in the mid-right part of Fig. 9.

4.6 Relational Aggregations on CPUs and GPUs

After a database tuple is associated with one or more keys based on spatial, temporal and spatiotemporal relations, the next step in spatial and temporal aggregations is to derive statistics in groups defined by the individual keys or their combinations which can be informative in decision making. Counting and similar relational aggregation operations in deriving group-based statistics are fundamental operators in relational databases and have been well researched, including some recent work on parallel aggregations on many-core GPUs [9][39][40]. A straightforward implementation of the relational aggregation operators on CPUs is to use STL map data structure to store (*key*, *stat*) pairs where *stat* can be one or more statistics, such as *count*/*sum*/*avg*/*min*/*max*. As the number of cores on multi-core CPUs is currently limited, a commonly used strategy on parallelizing the relational aggregation operations on multi-core CPUs is to provide each thread (typically assigned to a processor exclusively) a private copy of the map structure so that threads can work in parallel over non-overlapping partitions of tuples and avoid using expensive locks. The private map structures are then combined to derive the final result across threads/cores using a single thread. Unfortunately, this strategy is largely unrealistic on many-core GPUs as it is neither possible nor efficient to provide hundreds of thousands of threads with their private copies of result structures (e.g., maps). Furthermore, as the map structure is essentially a hashing based structure, it requires random accesses to hash tables and may incur significant cache misses and reduce performance. While large caches on CPUs can tolerate random data accesses to a certain extent, hashing typically performs poorly on GPUs due to the uncoalesced data accesses among threads in computing blocks [59].

Given that sorting is among the best studied parallel algorithms and efficient sorting implementations are provided by major parallel packages on parallel systems, we propose a *Transform-Sort-Reduce* based approach for relational aggregations on GPUs. The approach first derives keys based on domain knowledge, then sorts database tuples based on the keys and finally reduces on groups (identified by the keys) to produce the desired statistics for the groups. We note that many statistic operators, including the commonly used *count*/*sum*/*avg*/*min*/*max* operators that are supported by relational DBMS, are associative and can be applied in parallel within groups. Our GPU implementation of the relational aggregation module closely follows the three-step procedure design. For the first step in combining key attributes and generating keys for all tuples, the element-wise operation is highly parallelizable and is efficiently supported by several parallel libraries, including Thrust (using *transform* primitives). Note that the decision on whether to concatenate relevant value attributes to form a new relation (in a way similar to materialized views in relational databases) or to directly operate on the original relation is left to applications. As we have adopted a column-oriented database physical layout, deriving a new relation is performed by vertically concatenating relevant in-memory column files. For the second step in sorting the database tuples based on their keys, as radix sorting on fixed-length keys are typically more efficient than comparison based sorting and have a theoretical linear complexity with respect to number of data items to be sorted, our implementation requires

formulating keys as 32-bit or 64-bit integers to take the advantage of efficient parallel sorting implementations on GPUs. We argue that 32-bit keys can be sufficient for many practical applications. Since users are typically interested in only a limited amount of top-ranked groups, the keys can be hierarchically refined into multi levels while still using 32-bit keys at each level. For the third step in applying reduction operators within groups to derive statistics, Thrust provides a *reduce (by key)* primitive that meets our requirement.

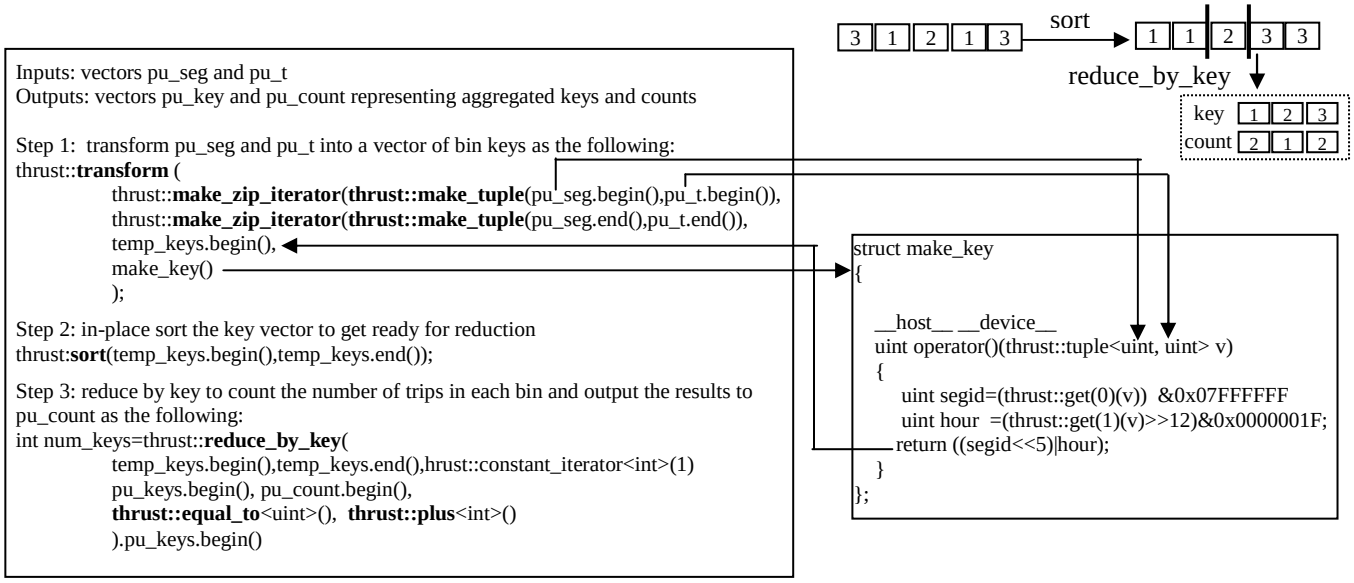

Fig. 10. Code Segment of a Parallel Relational Aggregation using Spatiotemporal Keys

Assuming that we want to perform a spatiotemporal aggregation on the street segment and hour, i.e., counting the number of taxi pickup locations at each of the street segments at the each of the 24 hours, Fig. 10 provides the code segment for illustration purposes. Here we are given two vectors with the first storing the street segments and the second storing the pickup times in the compressed form (Section 3.2) for all taxi trip records using a column-oriented data layout. Note that the *zip* iterator is used to combine the elements in the two input vectors into tuples in the *temp_keys* vector so that they can be used in the required functor (C++ function object) in the *transform* primitive to convert the segment identifiers and pickup hours into keys for reduction. The last five bits in a resulting key are allocated to hour ($24<2^5$) and the rest of the bits are allocated to segment identifiers which allows up to 2^{27} segment identifiers and is sufficient in our application. The same procedure can be applied when variables of higher dimensions are involved in key formations. The *reduce_by_key* primitive in Thrust is a segmented version of the regular *reduce* primitive. To help understand the procedure, a simple example is provided in the top-right part of Fig. 10. After sorting in Step 2, the same keys are arranged consecutively in the *temp_keys* vector. For each of the unique keys in the *temp_keys* vector (which is output to the *pu_keys* vector), the count in the *pu_count* vector is increased (defined by the *thrust::plus* functor) by 1 (as defined by the *thrust::constant_iterator*). The primitive allows us to define how to determine whether two keys are equal by providing a functor to replace the *thrust::equal_to* functor and how to perform the reduction by providing a functor to replace the *thrust::plus* functor and thus is very flexible. Since *thrust::make_tuple*

takes up to 10 parameters to make tuples, it should be more than sufficient to combine dimensional values and make keys in most cases.

The readers might have observed that the same *Transform-Sort-Reduce* based relational aggregation scheme can also be applied to multi-core CPUs. Indeed, parallel libraries on CPUs, such as Intel TBB, provide similar parallel primitives such as *parallel_for*, *parallel_sort* and *parallel_reduce* which make the multi-core CPU implementation of parallel relational aggregations possible. However, as shown in Section 5, sorting on multi-core CPUs is several times slower than sorting on GPUs in our experiment system which makes the primitive-based multi-core CPU implementation less preferable. On the other hand, while it is more efficient for GPUs to use more costly sorting operator to coordinate a large number of threads to make full use of the massively data parallel computing power of the hardware, multi-core CPUs are characterized by having small numbers of powerful processors with large caches. The *Transform-Sort-Reduce* scheme that works well for GPUs may not be the most efficient implementation for multi-core CPUs. In fact, as many of the relational aggregation operators are associative, it is unnecessary to produce a total order through sorting. As mentioned earlier, given that the numbers of cores on CPUs are typically small, when the numbers of groups are not extremely large, it is technically feasible to provide each CPU thread a private copy of data structures for storing statistics. The advantages of using private copies are to completely eliminate concurrent data accesses that typically require expensive locking, and, to eliminate expensive sorting. Each thread can just sequentially loops through the partition of the tuples assigned to it and generate statistics for the partition serially before the private copies of the statistics are combined. To reduce the memory management overheads in STL map structures, in our multi-core CPU implementation, we use dynamic allocated arrays instead as the sizes of keys are often known before relational aggregations. As shown in Section 5.5, the efficiency of relational aggregations can be significantly improved by using pre-allocated dynamic arrays and narrow the performance gaps between the GPU and CPU implementations.

We would like to take the opportunity to briefly comment on the implementations using native parallel programming languages or similar approaches (e.g., CUDA on GPUs and Pthreds on CPUs) and those that are based on parallel primitives (e.g., Thrust on GPUs and TBB on CPUs). In the context of OLAP aggregations, the work reported in [9] adopted a native implementation approach which requires a deep understanding of GPU hardware details and high parallel programming skills. Even though parallel reduction is a well-studied problem and the mapping between the reduction primitive and OLAP aggregations is relatively straightforward, it remains non-trivial to implement the aggregations with good performance using a native programming approach [9]. Another research effort on MOLAP cube [39] utilized parallel primitives which allowed the authors to focus on high-level constructs without diving into too many hardware details. While we are in favor of the parallel primitive based approach in general due to its simplicity and portability, the tradeoffs between code efficiency and coding complexity of parallel primitive based implementations and native implementations need to be justified in different applications. We also note that the primitive based implementation does incur some overheads that can be avoided if implemented directly on top of native parallel programming languages. For example, the *temp_keys* vector in Fig. 10 of our example is accessed multiple times when different primitives are invoked. In addition, most parallel primitives are designed for 1-dimensional data that can be represented as arrays or vectors. The similar arguments can be applied to CPU-based implementations in general, for example, Pthread and TBB. However, as multi-core CPUs typically favor coarse-grained task level parallelization and allow serial

implementation within parallel tasks, the differences on multi-core CPUs are not as significant as those on GPUs from an implementation perspective in the context of data management applications. To the best of our knowledge, there are no parallel libraries that support multi-dimensional data and provide a similar set of functionality. Our designs on parallel spatial indexing, filtering and refinements have the potential to be abstracted as multi-dimensional primitives for spatial and temporal aggregations and we leave this interesting topic for future work.

5 EXPERIMENTS AND RESULTS

5.1 Data and Experiment Setup

Through a partnership with the NYC Taxi and Limousine Commission (TLC), we have access to roughly 300 million GPS-based trip records collected during a period of about two years (2008-2010). In this study, we use the approximately 170 million pickup locations and times in 2009 for experiments. The NYC DCP LION street network dataset with 147,011 street segments is used as the urban infrastructure data to associate taxi locations with street segments and subsequent spatial and temporal aggregations. Among the 168,379,168 taxi pickup locations in NYC, the majority are successfully associated with their nearest street segments within R=250 feet. However, there are 867,163 (0.515%) locations whose computed shortest distances are more than 250 feet. They are considered as outliers and are excluded from subsequent analysis. All experiments are performed on a Dell Precision T5400 workstation equipped with dual quad-core CPUs running at 2.0 GHZ with 16 GB memory, a 500 GB hard drive and an Nvidia Quadra 6000 GPU device with 448 cores (running at 1.15 GHZ) and 6GB memory.

5.2 Results on MLQ-based Spatial Associations on GPUs

Fig. 11 shows the results of MLQ-based spatial associations implemented on GPUs using the maximum quadrant depth L=16 (2 feet cell resolution at the finest level), the maximum point quadrant size K= 512 and the expanded window width of R=25 feet for polyline MBBs. In this figure, the horizontal axis represents the months in the year 2009 that are used in the spatial associations, N1 is the number of point locations, N2 is the number of derived point quadrants, T1, T2, T3, and T4 are the runtimes measured in seconds of the four modules, point indexing, polyline indexing, spatial filtering, and spatial refinement, respectively, and T is the total runtime. We can see that the runtimes of point indexing dominate the total runtime in all tests and increase almost linearly with the number of point locations. This is mostly due to the radix-sort based parallel primitive for sorting that incurs linear runtimes with respect to the number of points. The runtimes of the remaining parallel primitives are mostly linear with respect to the number of points or quadrants, except in cases that a same point quadrant might be paired with multiple grid cells. We note that the runtimes for point indexing have already included data transfer times between CPUs and GPUs which account for nearly 25% of the end-to-end runtime using 12 months data (the whole year of 2009). Also note that the polyline indexing time is fixed across the months, which is relatively insignificant. From the results we can see that it takes about 15 seconds to complete the spatial associations between the 168.38 million pick-up locations and the 147 thousand street segments based on the nearest-neighbor searching principle within a 25 feet window width. The GPU-based spatial associations are considerably faster than running queries similar to query Q5 (listed in the Appendix at the end of the paper) in PostgreSQL that takes hours to days. Although it is not our intension to compare our results with PostgreSQL directly as they are designed for different purposes and exploit different sets of

technologies, the results clearly demonstrate the level of achievable performance on aggregating large-scale geospatial data on modern commodity parallel hardware.

An interesting observation of the experiment results on spatial filtering runtimes is that, as the number of point locations grows across months, the spatial filtering runtime actually decreases which is counter-intuitive. A careful examination reveals that, as the threshold K is fixed in our experiments, when the number of point locations is small, the number of derived point quadrants is likely to be large which results in a large number of cells after rasterization. Subsequently, it takes longer for binary searching, sorting and duplication removal. As the spatial filtering runtimes can be several times larger than the spatial refinement runtimes and even larger than the point indexing runtimes (for cases using smaller numbers of months in Fig. 11), it is desirable to reduce the runtimes as much as possible as users generally expect lower response times when the numbers of points are small. As such, we suggest to use adaptive grid resolutions in the GPU-based MLQ implementation, i.e., use coarser grid resolutions for smaller numbers of points being associated. Assuming that statistical distributions of point locations do not change significantly across months, it is likely that a cost model can be formally derived to suggest an optimal grid resolution for the implementation. This is left for our future work.

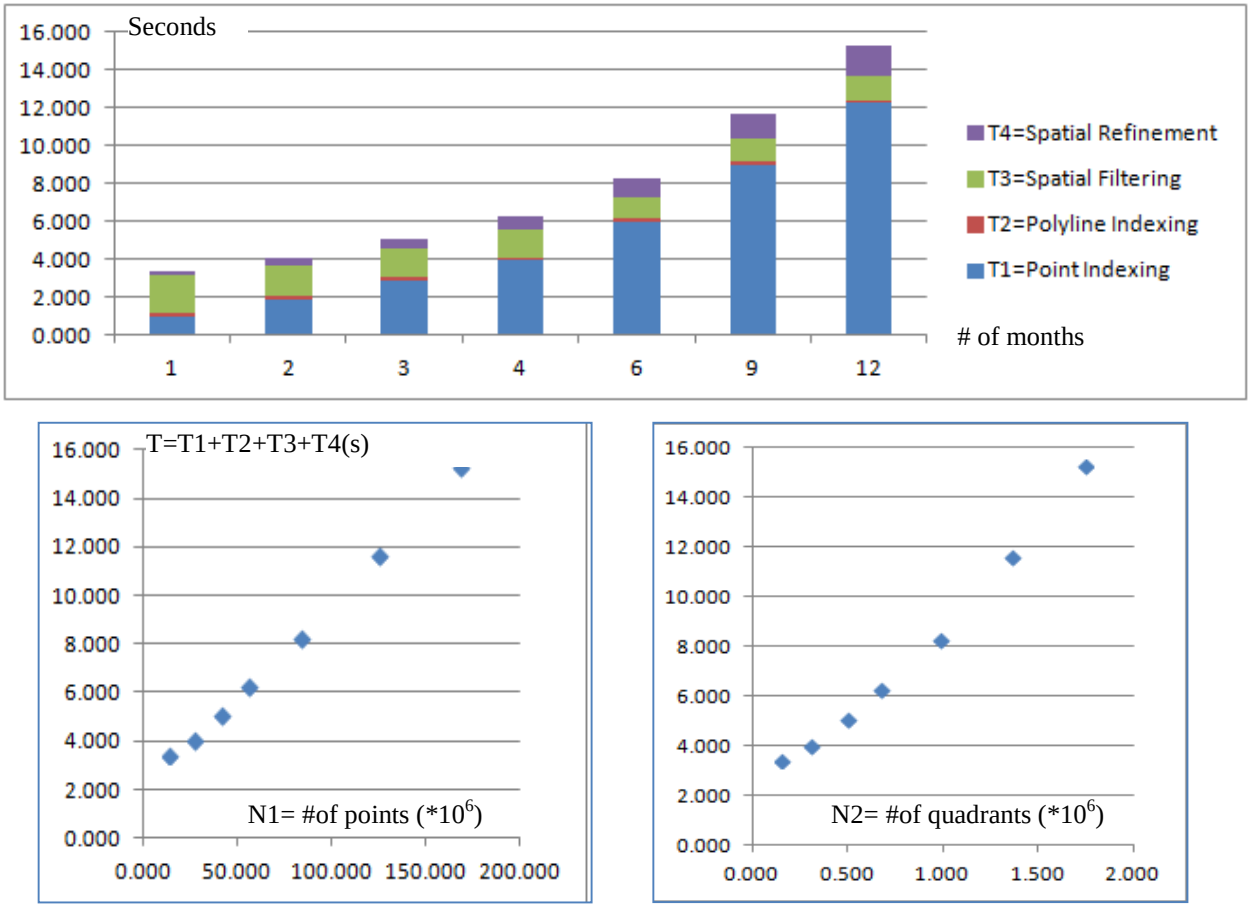

Fig. 11 Plot of Runtimes of MLQ-based Spatial Associations on GPUs

To further evaluate the performance of many-core GPUs for spatial associations, we have performed the same sets of operations corresponding to T1 (point indexing runtime) and T4 (spatial refinement runtime) on a single core CPU for the points in all the 12 months. The runtimes are T1'=162.004 seconds and T4'=35.338 seconds, respectively. We have not measured T2' (polyline indexing runtime) and T3' (spatial filtering runtime) on single core CPUs because we are not aware of efficient implementations of main-memory data structures for polyline indexing and spatial filtering. It would not be fair to simply port their GPU implementations to CPUs which might not be efficient. However, even if T2' and T3' are excluded, we still get a 13X speedup calculated as (T1'+T4')/(T1+T2+T3+T4)= 197.342/15.236. The result indicates the efficiency of GPU based spatial association based on the MLQ design.

5.3 Results on FSG-based Spatial Association on Multi-core CPUs and GPUs

We have tested our FSG-based spatial association module on multi-core CPUs under the following settings. A fixed grid cell size G=16 feet is used for all experiments. We have chosen to experiment on four *R* values (25 feet, 50 feet, 100 feet and 250 feet) to examine how end-to-end runtimes change as the polyline expanded window width increases. Using larger expanded window widths typically results in larger numbers of (cell, polyline) pairs that need to be refined after the spatial filtering. As such, both spatial filtering and spatial refinement runtimes are likely to increase. As indicated in Table 1 in Section 4, we use GNU Parallel Mode for sorting on multi-core CPUs. As the Polyline Indexing module is independent of FSG/MLQ designs, we re-use the MLQ implementation on GPUs directly and port it to multi-core CPUs through Thrust by changing the Thrust library device backend from CUDA on GPUs to TBB on multi-core CPUs. For spatial filtering, we have implemented the FSG-based design on GPUs to compare with MLQ based GPU implementation and port it to multi-core CPUs in a similar way. Also as discussed in Section 4.3, although we have ported the FSG based GPU implementations to multi-core CPUs (results listed in Table 2 in the FSG-MC column), we note that the library overheads for the TBB ports may be higher than the overheads for the native CUDA implementation of the Thrust library and the factor is taken into account during the analysis. While it is beyond the scope of our work to optimize the Thrust library on multi-core CPUs, we plan to natively implement the parallel primitives that are used in polyline indexing and spatial filtering based on the FSG design to improve their performance in our future work. For spatial refinement module using the FSG design, since we have explored VPUs in our multi-core CPU implementations, the runtimes for the following implementations are listed in Table 2: serial implementation using a single core (SC), parallel implementation on a single core with VPU (SC-VPU), parallel implementation on multiple cores without VPU (MC), and parallel implementation on multiple cores with VPU (MC-VPU). The runtimes for the spatial filtering module using the GPU implementations based on both FSG and MLQ designs under the four *R* values are also listed in Table 2 for comparison purposes.

From Table 2, we can see that the FSG-based multi-core CPU implementation for point indexing is about 2.3X faster than the MLQ-based GPU implementation (12.233/5.382). On the other hand, while the GPU implementation has achieved 13X speedups over serial implementation in the MLQ approach, only a 4.2X speedup (22.447/5.382) has been achieved for multi-core implementation over serial implementation in the FSG approach. This can be explained by that, as detailed in Section 4.1, the FSG design is significantly simpler than the MLQ design which subsequently requires much less computation. Unfortunately, the large memory footprints (for both the resulting grid cells and temporal memory during sorting) have prevented us from implementing the Point Indexing module on GPUs based on the FSG design

for direct comparisons. In addition, the CPU-based implementation of the FSG design does not require copying the point data from CPUs to GPUs which also contributes to its efficiency. Furthermore, as sorting is typically memory intensive and the bandwidth of our GPU device is much higher (144 GB/s vs. 2*12.8 GB/s), we would expect the implementation of the FSG design on GPUs to be more efficient than the multi-core GPU implementation when GPU memory capacity gets larger. We also suspect that insufficient memory bandwidth on multi-core CPUs is one of the important reasons that the multi-core CPU implementation achieves only half of the theoretical speedup (8X for 8 cores). As the future generation CPUs are likely to have higher memory bandwidths and more cores, we expect the performance of the multi-core CPU based implementation will increase accordingly.

Table 2. Results on FSG-based Spatial Associations

		FSG-SC	FSG-SC-VPU	FSG-MC	FSG-MC-VPU	FSG-GPU	MLQ-GPU
Point Indexing (PNI)		22.447		5.382			12.233
Polyline Indexing (PLI)	R=25			1.750			0.181
	R=50			2.448			0.244
	R=100			4.140			0.401
	R=250			13.450			1.148
Spatial Filtering (SF)	R=25			0.738		0.068	1.221
	R=50			0.984		0.083	1.391
	R=100			1.521		0.118	1.702
	R=250			4.305		0.264	3.207
Spatial Refinement (SR)	R=25	21.126	7.251	2.631	0.930	1.002	1.601
	R=50	28.642	9.480	3.559	1.198	1.267	*2.088*
	R=100	43.419	13.913	5.440	1.749	1.839	3.019
	R=250	100.738	31.117	12.487	3.874	4.156	6.573

For polyline indexing, the runtimes under all *R* values are relatively insignificant on GPUs. However, their runtimes are more than 10X higher when the GPU implementation (the same in both the MLQ and FSG designs) is ported to multi-core CPUs. While the low performance can also occur due to the small number of cores and low memory bandwidth as we have discussed previously (and low CPU clock rate in our experiment system), we also believe that the Thrust library overheads of the TBB port on multi-core CPUs are significantly higher than its native implementation on GPUs. For future work, we plan to provide an optimized native implementation of the polyline indexing module in order to compare the best performance of the multi-core CPU implementation with the performance of the GPU implementation.

The Spatial Filtering results provided in Table 2 show that the FSG-based GPU implementation is 10-20X faster (by dividing values in the MLQ-GPU column by those in the FSG-GPU column in the four SF rows) than the MLQ-based GPU implementation which is preferred. When comparing the FSG-based implementation on multi-core CPUs and GPUs (using the values in FSG-GPU column and FSG-MC column in the same SF rows), we can see that the GPU-based implementation is again more than 10X faster than the multi-core CPU implementation by porting the same implementation from CUDA to TBB. Again, we attribute

this result to a small number of cores and low memory bandwidth on multi-core CPUs in our experiment system and the higher library overheads on CPUs.

More comprehensive comparisons can be performed for the Spatial Refinement module as all the six implementations are available for this module. Our results in Table 2 show that the runtimes of the MLQ-GPU implementation are approximately 60% higher than the runtimes for the FSG-GPU implementation for the four R values. This may also indicate that load balancing is not a dominating factor due to fine data decompositions as discussed in Section 4. It might be more interesting to compare the performance of the five implementations for the FSG design, especially the FSG-MC-VPU implementation, which has the best performance on multi-core CPUs, and the GPU implementation. From Table 2 we can see that the GPU implementation has achieved 13-15X speedups over the serial implementation in the FSG approach for the four *R* values (FSG-SC/FSG-GPU for SR rows). They are comparable with the speedups of the GPU implementation over the serial implementation in the MLQ based approach (13X for *R*=25 as reported in Section 5.2). When comparing the GPU performance (FSG-GPU) and multi-core CPU performance (FSG-MC), we can see that the GPU implementation has achieved 2.6-3.0X speedups over multi-core CPUs (FSG-MC/FSG-GPU for SR rows). When comparing the runtimes of FSG-MC with those of FSG-SC, we observe linear speedups (around 8X) and we attribute these results to both the parallel design with regular data access patterns and the efficiency of TBB scheduling in balancing the workloads among grid cells. We next turn to analyzing the CPU performance with multi-cores and VPUs.

When comparing the runtimes between the FSG-SC and FSG-SC-VPU columns in Table 2 for the spatial refinement module, we can see that the speedups due to VPUs (vectorization) are around 3X (FSG-SC/FSG-SC-VPU for SR rows). Slightly lower speedups (but still around 3X) can be observed by comparing the runtimes in the FSG-MC and FSG-MC-VPU columns where all cores are utilized. Given that the VPUs on the Intel Xeon CPUs in our experiment system have 128-bit SIMD width (4-way) and the maximum possible speedup is only 4X, the achieved speedups are very significant, especially considering that not all the numbers of points in grid cells can be divided by 4 and the remaining points that cannot be fed to a SIMD batch need to be processed sequentially. By combining multi-cores and VPUs, the multi-core CPU implementation has achieved 21-24X speedups over serial implementation without using multi-cores and VPUs (FSG-SC/FSG-MC-VPU for SR rows). The performance is about 5%-7% better than the GPU implementation in the FSG design (calculated as (FSG-GPU/FSG-MC-VPU)-1 for SR rows). As the CPUs in our experiment system are relatively weak (released in 2007), we believe more recent systems with more CPU cores and higher CPU frequencies can achieve better performance. The results are exciting in the sense that, when parallel hardware (including multi-cores and VPUs) is fully utilized, multi-core CPUs have the potential to achieve comparable or even better performance than GPUs in our application.

5.4 Results on End-to-End Response Times of Spatial Associations on Hybrid Systems

While we have compared the performance of the individual modules on either multi-core CPUs or GPUs or both under different settings for the FSG-based approach in Section 5.3, we next provide comparisons on the end-to-end response times of spatial associations in this subsection. Although the end-to-end response time of the MLQ-based approach for *R*=25 feet has been provided and discussed in Section 5.2, we would like to compare the two approaches under different *R* values. In Table 3, we refer the MLQ-based approach as *Config1* where all the runtimes are based on the GPU implementations. In contrast, for *Config2*, the runtimes are mixtures of the GPU and CPU implementations by taking the best performance of all available

implementations of the respective module. The decision is based on several considerations. First, only CPU implementation is available for the FSG design as the point data is beyond the memory capacity of the GPU device in our experiment system for the GPU implementation using the FSG design. Second, the multi-core CPU implementations of the polyline indexing and spatial filtering modules are naïve ports of the corresponding GPU designs and are not optimized. It would be unfair to use them in calculating the end-to-end response times for multi-core CPUs. Third, by utilizing VPUs on CPUs, the CPU based implementation of the Spatial Refinement module actually slightly outperforms the corresponding GPU implementation although the GPU implementation is still about 2.6-3.0X better than the multi-core CPU implementation without VPUs. It would not be appropriate to simply refer to the VPU enhanced implementation as a multi-core CPU implementation. By taking the best runtimes of the modules in the FSG-based approach, we can compute the best performance in the hybrid CPU-GPU system for the FSG-based approach and compare it with the MLQ-based approach with a pure GPU implementation.

Table 3. Runtimes of Multiple Design and Implementation Configurations

		R=25	R=50	R=100	R=250
Config1 (MLQ-GPU)	PNI				12.233
	PLI	0.181	0.244	0.401	1.148
	SF	1.221	1.391	1.702	3.207
	SR	1.601	*2.088*	3.019	6.573
	Tot	15.236	15.956	17.355	23.161
Config2 (FSG-Hybrid)	PNI(FSG-MC)				5.382
	PLI (FSG-GPU)	0.181	0.244	0.401	1.148
	SF (FSG-SC)	0.068	0.083	0.118	0.264
	SR (FSG-MC-VPU)	0.930	1.198	1.749	3.874
	Tot	6.561	6.907	7.65	10.668
Speedup=Config1-Tot/Config2-Tot		2.32	2.31	2.27	2.17

From the last row of Table 3 we can see that the FSG-hybrid configuration has achieved a little over 2X speedups over the MLQ-GPU configuration which indicates that the FSG design is a better choice for spatial associations on our experiment system. We expect that, after the native multi-core CPU implementations of the polyline indexing and spatial filtering modules become available, although it is likely to increase the end-to-end runtimes of the two modules when compared to the current GPU implementations, the FSG implementation that runs completely on multi-core CPUs will still likely be better than the MLQ-based approach on GPUs. The better performance of the FSG approach on multi-core CPUs can be attributed to three major factors. The first factor is the algorithmic improvement to significantly reduce the computing overheads of spatial filtering/refinement as well as point indexing. The efficiency on reducing computing overheads is evidenced by comparing the runtimes in FSG-GPU and MLQ-GPU columns in Table 2 where GPU implementations for both approaches are available. The efficiency on point indexing based on the FSG design can be inferred based on the fact that sorting the 168 million records should take less than 1 second on the GPU if memory is sufficient based on previous results [55] while it is more than 12 seconds based on the MLQ design. The second factor is the elimination of the need to transfer data between CPUs and GPUs (3 seconds based on our

experiments). The third factor is the effective utilization of VPUs on CPUs that have been largely untouched in previous studies in the spatial refinement module. We expect the end-to-end runtimes of the pure multi-core CPU based implementation can be further reduced after utilizing the SIMD processing powers on VPUs in other three modules in spatial associations. Finally, despite the fact that different parallel designs and implementations have resulted in different performance, we would like to note that the end-to-end runtimes of both the FSG-Hybrid and MLQ-GPU implementations are in the order of 5-25 seconds for the *R* values ranging from 25 feet to 250 feet as shown in Table 3. The results clearly indicate the high efficiency of our parallel designs and implementations although there are still rooms for improvements. The end-to-end response times are getting closer to meeting the requirement of interactive spatial OLAP query processing. We believe interactive OLAP queries on spatially associating hundreds of millions of points and hundreds of thousands of polylines are technically feasible through further algorithmic engineering, implementation optimization and heterogeneous/distributed processing and they are left for our future work.

5.5 Results on Parallel Relational Aggregations

After spatial and temporal associations, spatial and temporal aggregations are reduced to relational aggregations. We have performed three groups of experiments on relational aggregations on both many-core GPUs and multi-core CPUs using the 168 million taxi pick up locations after spatial and temporal associations. The first group of experiments counts on the 147,011 street segments after spatial aggregations based on the nearest neighbor search. The second group of experiments counts on the 24 hours (temporal aggregation) and the third group of experiments counts on both street segments and hours (spatiotemporal aggregation). Clearly, the number of bins in the spatial aggregations based on segment identifiers is 3-4 orders of magnitude larger than the number of bins in the temporal aggregations based on hours. They can be used to represent the two extremes in aggregations with respect to cardinality. The experiment results for the three groups of queries on multi-core CPUs and many-core GPUs are plotted in Fig. 12 and are compared with serial implementations. For the CPU implementations (serial and multi-core), we have experimented on using both STL map structures and dynamic arrays. For the GPU implementation, as it is based on the Thrust parallel library, we use multiple vectors instead.

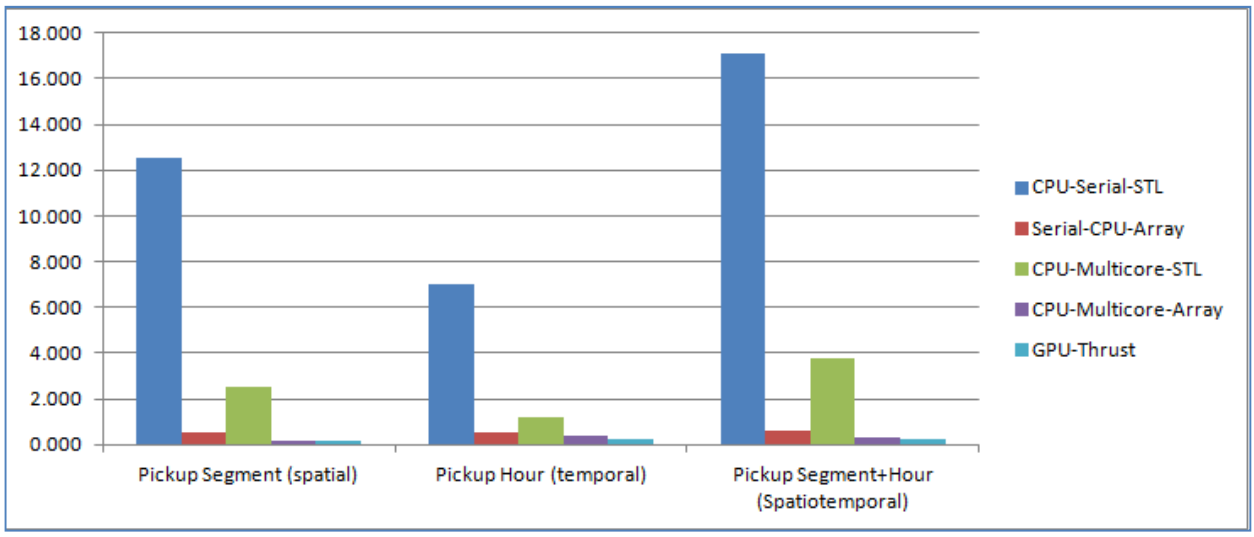

Fig. 12. Runtimes of Parallel Relational Aggregations

Our results show that the speedups of the multi-core CPU implementations over the serial implementations range from 4.5-6.0X when STL maps are used and 1.5-2.7X when dynamic arrays are used. The results are expected as the relational aggregations are mostly memory intensive which may severely limit the achievable speedups (maximum 8X for 8 cores). When dynamic arrays are used, memory bandwidth contention is likely to be a major factor in further limiting the achievable speedups. The GPU implementations have achieved slightly better performance than the multi-core implementations using dynamic arrays. Given that the GPU device used in our experiment system has a larger number of processing cores (although weaker than CPU cores) and higher memory bandwidth, we believe it is quite possible to further improve the performance of the GPU implementation by using CUDA directly to reduce the Thrust library overheads. While it remains nontrivial to design and implement sophisticated parallel primitives that are optimized for OLAP applications using low-level programming languages and libraries on parallel hardware, we plan to provide designs and implementations that can be compatible on both GPUs and VPUs of CPUs and make full use of their SIMD parallel processing power.

6 CONCLUSION AND FUTURE WORK

In this study, we reported our designs, implementations and experiments on developing a data management platform and a set of parallel techniques to support high-performance online spatial and temporal aggregations on multi-core CPUs and many-core GPUs that are becoming increasingly available but largely under-utilized for OLAP applications. Our results have shown that we are able to spatially associate nearly 170 million taxi pickup location points with their nearest street segments among 147,011 candidates in about 5-25 seconds, depending on different configurations of indexing structures, computing models and parameters such as expanded window widths. After spatially associating points with road segments, spatial, temporal and spatiotemporal aggregations are reduced to relational aggregations and can be processed in the order of a fraction of a second on both multi-core CPUs and GPUs. The experiment results support the feasibility of building a high-performance OLAP system for processing large-scale taxi trip data for real-time, interactive data explorations on GPUs, multi-core CPUs and their hybridizations.

For the future work, first of all, we would like to further reduce the processing times for both spatial associations and relational aggregations by fine tuning important parameters and further reducing memory footprint. Second, to ensure usability, we would like to investigate the appropriate spatial and temporal resolutions so that interactive OLAP processing can be smoothly performed on commodity personal computers with different hardware configurations. Third, while our designs and implementations reported in this study are application driven, we are interested in formally analyzing the complexity and scalability of the proposed solution by varying numbers of CPU/GPU processors and empirically validating the analysis through extensive experiments. Fourth, while our techniques are designed and developed mostly in the context of managing large-scale OD data, many of them are applicable to other types of spatial and spatiotemporal data and we plan to investigate the possibilities. Finally, we plan to explore cluster computing technologies to process larger scale data, for example, multi-year and multi-city taxi trip data and cell phone call log data.

ACKNOWLEDGEMENT

This work is supported in part through PSC-CUNY Grants #65692-00 43 and #66724-00 44 and NSF Grants IIS-1302423 and IIS-1302439. We thank Dr. Camille Kamga at CUNY City College for providing the NYC taxi trip data and Dr. Hongmian Gong at CUNY Hunter College for insightful discussions on urban applications of GPS data.

REFERENCES

1. J. Reades, F. Calabrese, A. Sevtsuk, C. Ratti, Cellular census: Explorations in urban data collection, IEEE Pervasive Computing 6 (3) (2007) 30-38.
2. F. Calabrese, M. Colonna, P. Lovisolo, D. Parata, C. Ratti, Real-time urban monitoring using cell phones: A case study in Rome, IEEE Transactions on Intelligent Transportation Systems 12 (1) (2011) 141-151.
3. M. A. Vasconcelos, S. Ricci, J. Almeida, F. Benevenuto, V. Almeida, Tips, dones and todos: uncovering user profiles in Foursquare, in: Proceedings of the fifth ACM international conference on Web Search and Data mining, WSDM'12, 2012, pp. 653-662.
4. F. Calabrese, K. Kloeckl, WikiCity: Real-time location-sensitive tools for the city, in: M. Foth (Ed.), Handbook of Research on Urban Informatics: The Practice and Promise of the Real-Time City, IGI Global, 2008, pp.390-413.
5. G. Friedland, J. Choi, A. Janin, Video2GPS: a demo of multimodal location estimation on Flickr videos, in: Proceedings of the 19th ACM international conference on Multimedia, MM'11, 2011, pp. 833-834.
6. J. Zhang, C. Kamga, H. Gong, L. Gruenwald, U2SOD-DB: a database system to manage large-scale ubiquitous urban sensing origin-destination data, in: Proceedings of the ACM SIGKDD International Workshop on Urban Computing, UrbComp'12, 2012, pp. 163-171.
7. S. Shekhar, C. Chawla, Spatial Databases: A Tour, Prentice Hall, 2003.
8. C. Duntgen, T. Behr, R. Guting, BerlinMOD: a benchmark for moving object databases, The VLDB Journal 18 (2009) 1335-1368.
9. T. Lauer, A. Datta, Z. Khadikov, C. Anselm, Exploring graphics processing units as parallel coprocessors for online aggregation, in: Proceedings of the ACM 13th international workshop on Data warehousing and OLAP, DOLAP '10, 2010, pp. 77-84.
10. R. Wrembel, Data warehouse performance: Selected techniques and data structures, in: M.-A. Aufaure, E. Zimnyi (Eds.), Business Intelligence, Vol. 96 of Lecture Notes in Business Information Processing, 2012, pp.27-62.
11. J. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach (5th ed.), Morgan Kaufmann, 2011.
12. E. H. Jacox, H. Samet, Spatial join techniques, ACM Transaction on Database Systems 32 (1) (2007) 7-24
13. Y. Zheng, Y. Liu, J. Yuan, X. Xie, Urban computing with taxicabs, in: Proceedings of the 13th international conference on Ubiquitous Computing, UbiComp'11, 2011, pp. 89-98.
14. S. Phithakkitnukoon, T. Horanont, G. D. Lorenzo, R. Shibasaki, C. Ratti, Activity-aware map: Identifying human daily activity pattern using mobile phone data, in: Proceedings of the First International Conference on Human Behavior Understanding, HBU'10, 2010, pp. 14-25.
15. Y. Zheng, Y. Chen, X. Xie, W.-Y. Ma, Geolife2.0: A location-based social networking service, in: Proceedings of the 10th International Conference on Mobile Data Management: Systems, Services and Middleware, MDM'09, 2009, pp. 357-358.

16. J. Yuan, Y. Zheng, L. Zhang, X. Xie, G. Sun, Where to find my next passenger, in: Proceedings of the 13th international conference on Ubiquitous Computing, UbiComp'11, 2011, pp. 109-118.
17. R. Xie, Y. Ji, Y. Yue, X. Zuo, Mining individual mobility patterns from mobile phone data, in: Proceedings of the 2011 international workshop on Trajectory Data Mining and Analysis, TDMA'11, 2011, pp. 37-44.
18. A. Vaisman, E. Zimanyi, What is spatio-temporal data warehousing?, in: Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery, DaWaK'09, 2009, pp. 9-23.
19. S. Rivest, Y. Bdard, P. March, Towards better support for spatial decision making: defining the characteristics, in: Proceedings of Geomatica, 2001, pp. 539-555.
20. A. O. Mendelzon, A. A. Vaisman, Temporal queries in OLAP, in: Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00, 2000, pp. 242-253.
21. R. do Nascimento Fidalgo, V. C. Times, J. da Silva, F. da Fonseca deSouza, GeoDWFrame: A framework for guiding the design of geographical dimensional schemas, in: Proceedings of the 6th International Conference on Data Warehousing and Knowledge Discovery, DaWaK'04, 2004, pp. 26-37.
22. K. Boulil, S. Bimonte, H. Mahboubi, F. Pinet, Towards the definition of spatial data warehouses integrity constraints with spatial OCL, in: Proceedings of the ACM 13th international workshop on Data warehousing and OLAP, DOLAP '10, 2010, pp. 31-36.
23. L. I. Gomez, S. Haesevoets, B. Kuijpers, A. A. Vaisman, Spatial aggregation: Data model and implementation, Information System 34 (6) (2009) 551-576.
24. A. Escribano, L. I. Gomez, B. Kuijpers, A. A. Vaisman, Piet: a GIS-OLAP implementation, in: Proceedings of the ACM 10th international workshop on Data warehousing and OLAP, 2007, pp. 73-80.
25. S. Bimonte, A. Tchounikine, M. Miquel, Spatial OLAP: Open Issues and a Web Based Prototype, in: Proceedings of the 10th AGILE International Conference on Geographic Information Science, 2007, pp. 1-11.
26. M. Scotch, B. Parmanto, Sovat: Spatial olap visualization and analysis tool, in: Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 2005.
27. O. Glorio, J.-N. Mazon, I. Garrigos, J. Trujillo, Using web-based personalization on spatial data warehouses, in: Proceedings of the 2010 EDBT/ICDT Workshops, EDBT '10, 2010, pp. 8:1-8:8.
28. T. L. L. Siqueira, C. D. de Aguiar Ciferri, V. C. Times, R. R. Ciferri, The SB-index and the HSB-index: efficient indices for spatial data warehouses, GeoInformatica 16 (1) (2012) 165-205.
29. J. J. Brito, T. L. L. Siqueira, V. C. Times, R. R. Ciferri, C. D. de Ciferri, Efficient processing of drill-across queries over geographic data warehouses, in: Proceedings of the 13th international conference on Data Warehousing and Knowledge Discovery, DaWaK'11, 2011, pp. 152-166.
30. K. Choi, W.-S. Luk, Processing aggregate queries on spatial OLAP data, in: Proceedings of the 10th international conference on Data Warehousing and Knowledge Discovery, DaWaK '08, 2008, pp. 125-134.

31. D. Papadias, P. Kalnis, J. Zhang, Y. Tao, Efficient OLAP operations in spatial data warehouses, in: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases, SSTD '01, 2001, pp. 443-459.
32. R. Obe, L. Hsu, PostGIS in Action, Manning Publications, 2011.
33. J. Zhang, S. You, L. Gruenwald, High-performance online spatial and temporal aggregations on multi-core CPUs and many-core GPUs, in: Proceedings of the ACM 15th international workshop on Data warehousing and OLAP, 2012, pp. 89-96.
34. Zhang, S. You, Speeding up large-scale point-in-polygon test based spatial join on GPUs, in: Proceedings of the ACM SIGSPATIAL Workshop on BigSpatial, 2012, 23-32.
35. J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufmann, 2013.
36. D. B. Kirk, W.-M. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach (2nd ed.), Morgan Kaufmann, 2012.
37. K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, B. Moon, Parallel data processing with MapReduce: a survey, SIGMOD Record 40 (4) (2012) 11-20.
38. M. Grund, J. Kruger, H. Plattner, A. Zeier, P. Cudre-Mauroux, S. Madden, Hyrise: a main memory hybrid storage engine, Proceedings of VLDB Endowment 4 (2) (2010) 105-116.
39. K. Kaczmarski, T. Rudny, MOLAP cube based on parallel scan algorithm, in: Proceedings of the 15th international conference on Advances in Databases and Information Systems, ADBIS'11, 2011, pp. 125-138.
40. E. A. Sitaridi, K. A. Ross, Ameliorating memory contention of OLAP operators on GPU processors, in: Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN '12, 2012, pp.39-47.
41. B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, P. V. Sander, Relational query coprocessing on graphics processors, ACM Transaction on Database Systems 34 (4).
42. M. McCool, J. Reinders, J. Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012.
43. Nvidia, Compute unified device architecture (CUDA), http://www.nvidia.com/object/cuda_home_new.html.
44. Intel, Thread building blocks (TBB), http://threadingbuildingblocks.org/.
45. J. Zhang, A brief introduction to parallel primitives, http://www-cs.ccny.cuny.edu/~jzhang/intro_pp.pdf.
46. J. Singler, B. Konsik, The GNU libstdc++ parallel mode: software engineering considerations, in: Proceedings of the 1st International Workshop on Multicore Software Engineering, IWMSE'08, 2008.
47. T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, J. Schaffner, SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units, Proceedings of the VLDB Endowment 2 (1) (2009) 385-394.
48. C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, P. Dubey, FAST: fast architecture sensitive tree search on modern cpus and gpus, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, SIGMOD '10, 2010, pp. 339-350.
49. T. Yamamuro, M. Onizuka, T. Hitaka, M. Yamamuro, VAST-Tree: a vector-advanced and compressed structure for massive data tree traversal, in: Proceedings of the 15th

International Conference on Extending Database Technology, EDBT '12, 2012, pp. 396-407.
50. M. Pharr, W. Mark, ispc: A SPMD compiler for high-performance CPU programming, in: Proceedings of Innovative Parallel Computing, InPar, 2012.
51. Metropolitan Transportation Authority (MTA), Subway and bus ridership,http://www.mta.info/nyct/facts/ridership/.
52. NYC Department of City Planning(DCP), DCPLION geographic base file of New York City streets, http://www.nyc.gov/html/dcp/html/bytes/dwnlion.shtml.
53. I. F. Vega Lopez, R. T. Snodgrass, B. Moon, Spatiotemporal aggregate computation: a survey, Knowledge and Data Engineering, IEEE Transactions on 17 (2) (2005) 271-286.
54. Wikipedia, Standard Template Library (STL), http://en.wikipedia.org/wiki/Standard_Template_Library.
55. D. Merrill, A. S. Grimshaw, High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing, Parallel Processing Letters 21 (2) (2011) 245-272.
56. N. Leischner, V. Osipov, P. Sanders, GPU sample sort, in: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS'10, 2010, pp. 1-10.
57. H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc., 2005.
58. M. F. Mokbel, W. G. Aref, Irregularity in high-dimensional space filing curves, Distributed and Parallel Databases 29 (3) (2011) 217-238.
59. D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, N. Amenta, Real-time parallel hashing on the GPU, ACM Transactions on Graphics 28 (5) (2009) #9.

APPENDIX SQL STATEMENTS FOR SPATIAL/TEMPORAL AGGREGATIONS IN POSTGRESQL

```
Q1: UPDATE t SET PUGeo = ST_SetSRID(ST_Point("PULong","PuLat"),4326);
Q2: UPDATE t SET DOGeo = ST_SetSRID(ST_Point("DOLong","DOLat"),4326);
Q3: CREAT INDEX ti_pugeo ON t USING GIST (PUGeo);
Q4: CREAT INDEX ti_dogeo ON t USING GIST (DOGeo);
Q5: SELECT DISTINCT ON (ID, PUT) ID, PUT, segmentid,
ST_Distance ( ST_Transform (PUGeo,2263), the_geom) as ndis INTO temp_PU FROM t, n
WHERE ST_DWithin (ST_Transform (PUGeo, 2263), the_geom, 100) ORDER BY PUT, ID, ndis
Q6: UPDATE t set PUSeg=(SELECT segmentid From temp_PU WHERE t.ID=temp_PU.ID AND t.PUT=temp_PU.PUT;
Q7: SELECT DISTINCT ON (ID, DOT) ID, DOT, segmentid,
ST_Distance ( ST_Transform (DOGeo,2263), the_geom) as ndis INTO temp_DO FROM t, n
WHERE ST_DWithin(ST_Transform(DOGeo,2263), the_geom, 100) ORDER BY DOT, ID, ndis
Q8: UPDATE t set DOSeg=(SELECT segmentid From temp_DO WHERE t.ID=temp_DO.ID AND t.DOT=temp_DO.DOT;
Q9: CREAT INDEX ti_pus ON t(PUSeg);
Q10: CREAT INDEX ti_dos ON t(DOSeg);
Q11: SELECT PUSeg, COUNT(*) FROM t GROUP BY PUSeg  ORDER BY PUSeg;
Q12: SELECT DOSeg, COUNT(*) FROM t GROUP BY DOSeg  ORDER BY DOSeg;
Q13: CREAT INDEX ti_put ON t (PUT);
Q14: CREAT INDEX ti_dot ON t (DOT);
Q15: SELECT EXTRACT (hour FROM PUT) as hour, count(*) FROM t GROUP BY hour ORDER BY hour
Q16: SELECT EXTRACT (hour FROM DOT) as hour, count(*) FROM t GROUP BY hour ORDER BY hour
```