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

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

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

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

ABSTRACT

Motivated by the practical needs for efficiently processing large-scale taxi trip data, we have developed techniques for high performance online spatial, temporal and spatiotemporal aggregations. These techniques include timestamp compression to reduce memory footprint, simple linear data structures for efficient in-memory scans and utilization of massively data parallel GPU accelerations for spatial joins. Our experiments have shown that the combined performance boosting techniques are able to perform various spatial, temporal and spatiotemporal aggregations on hundreds of millions of taxi trips in the order of a few seconds using commodity personal computers equipped with multi-core CPUs and many-core GPUs. The high throughputs in a personal computing environment are encouraging in the sense that high-performance OLAP queries on large-scale data is feasible when the parallel processing power of modern commodity hardware is fully utilized which is important for interactive OLAP applications.

1. INTRODUCTION

Multidimensional aggregation is considered as one of the most important computational building block in Online Analytical Processing (OLAP) [1]. Considerable works on developing efficient data structures and algorithms have been proposed for multidimensional aggregations on CPU uniprocessors in the past few decades [2]. Modern hardware architectures increasingly rely on parallel technologies to increase the processing power due to various limits in improving the speeds of uniprocessors [3]. 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 GPUs [3]. Despite the fact that parallel hardware are already available in the majority of commodity computers, there are still relatively few works 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. For example, counting the number of taxi pickups at each of the community districts or census blocks (spatial aggregation) and generating hourly histogram of drop-offs near the JFK airport (temporal aggregation).

In this study, we report our work on aggregating hundreds of millions of taxi trip records in the New York City (NYC) area which amount to approximately half a million taxi trip records in a day on average. Using the approximately 170 million taxi trip records in 2009, each of which includes a pickup location/time, a drop-off location/time and several additional data attributes such as fare, trip distance and duration, we have experimented various spatial and temporal aggregations on both multi-core CPUs and many-core GPUs. By utilizing spatial joins to support efficient online processing, we are able to achieve real-time responses for spatial, temporal and spatiotemporal aggregations at the 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 a few days to a few seconds. This makes it possible for urban geographers and transportation researchers to explore the large-scale taxi trip data online in an interactive manner.

The rest of the paper is arranged as following. Section 2 introduces the background, motivation and related works. Section 3 presents the spatial and temporal hierarchies in aggregating taxi trip data. Section 4 provides the details on the parallel implementations of the 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

Almost all taxi cabs in cities of the developed countries have been equipped with GPS devices and different types of trip related information are recorded. For example, the more than 13,000 GPS-equipped medallion taxicabs in the New York City (NYC) generate nearly half a million taxi trips per day and 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 [4]. Taxi trips play important roles in everyday lives of residents and visitors of NYC as well as any major city worldwide. Exploring traffic and travel patterns from large-scale taxi trip records is important in understanding human mobility and facilitating transportation planning.

OLAP technologies are attractive to explore the possible patterns from the large-scale taxi trip records. 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 spatio-temporal data which requires synergizing existing research on Spatial OLAP (or SOLAP) [5][6] and temporal OLAP (TOLAP) [5][7] and their combinations [5]. Due to the popularity of geo-reference data, there are increasing research and application interests in SOLAP. However, most of them focus on data modeling [5, 8], query languages [5, 10] and applications on top of spatial databases and Geographical Information System (GIS) [11-14]. 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 [2, 15-18]. SOLAP 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 our application given the large number of data records. We also note that the existing research on SOLAP mostly targeted at the traditional computing framework, i.e., disk resident data on uniprocessors based on

serial algorithms, which makes them 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 the hundreds of millions of record, 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 [19]. 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 *lion09C* 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 9.0, Q1 (|t|~=170 million and |lion09c|~=150 thousand) took 105.8 hours and Q5 took 34.43 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. As a matter of fact, we had decided to seek alternative solutions before all the 16 queries could complete during a reasonably long period because of the poor performance.

While we are aware that certain optimization techniques, such as setting proper parameters and data partitioning, can potentially improve the overall performance, we have concluded that OLAP queries based on traditional database systems can not achieve the performance level that we are aiming at for the data at the scale. Our additional experiment results have also revealed that the performance can be drastically improved by utilizing large main-memory capacities and GPU parallel processing [20][21][22]. 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 have already been equipped in commodity personal computers.

We refer to [1] for a brief review on parallel OLAP computation. We note that existing works on parallel OLAP mostly focused on parallelization on shared-nothing architectures while leaving parallelization on shared-memory SMP (Symmetric multiprocessing) architectures, including both multi-core CPUs and many-core GPUs, largely untouched. The number of processing cores on both single-node CPUs and GPUs are fast increasing. The mainstream Intel CPUs and Nvidia GPUs have 12 and 512 cores, respectively. Devices based the Intel Many Integrated Coe (MIC) architecture will soon be available on the market with 48 or more cores [23] and devices based on Nvidia Kepler architecture equipped with more than 3,000 cores [24] is currently available in the market. These inexpensive devices based on the 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 [25]. 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 multicore CPUs and GPUs for OLAP queries including aggregations. The design and implementation of the HYRISE system [26] has 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 [2][27][28][29] with demonstrable performance speedups. 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 [1][27], the number of data records in our application is almost two orders larger which makes GPU implementation more technically challenging.

3 SPATIAL AND TEMPORAL AGGREGATIONS OF TAXI TRIPS

The raw taxi service data has a few dozens of attributes and ten are considered most relevant to our analysis: driver identifier (ID), pick-up time (PUT), drop-off time (DOT), trip distance (DIS), trip fare (FARE), trip tip (TIP), pick-up latitude (PULat), pickup longitude (PULong) , drop-off latitude (DOLat) and drop-off longitude (DOLong). Various aggregations can be performed based on these data attributes and the planned support for spatial and temporal aggregations has been outlined in [20]. One of the primary focuses of this paper is to discuss the design and implementation details of these aggregations. For the sake of being self-content, we have replicated the design in Fig. 1 which will be discussed in detail in the following subsections. As spatial aggregations using uniform grids have been discussed in [20] in details and they belong to a different set of technologies, we have excluded them from the figure.

3.1 Spatial Associations and Aggregations

While the temporal aggregation hierarchy listed in Fig. 1 is relatively straightforward, more knowledge on the study area and application background is needed to understand the presented spatial hierarchy listed in the middle-left of the figure. There are three types of urban infrastructure data involved in our application, i.e., street networks (polyline), collectively exhaustive polygons and non- collectively exhaustive polygons. The distinction between the last two depends on whether the union of the polygons in the respective dataset covers the study area completely. Many administrative zones (such as community districts and police precincts) and census blocks/tracts belong to the first category. In this case, taxi trip locations can be associated with the polygons through point-in-polygon test, i.e., based on topological relationships [20][21]. For tax lots/blocks, they

represent the land parcels that are owned by individuals or institutions and do not cover road beds. As such, the association between taxi trip locations and the polygons is based on geometrical distances, e.g., nearest neighbor principle [20][22]. For the street network, we use the LION dataset published by the NYC Department of City Planning (DCP) [30] where street segments are nicely associated with quite a few types of polygon zones as shown in Fig. 1. To associate point locations with community districts or police precincts, while it is possible to spatially associate points with polygons though the point-in-polygon tests directly [21], if a point location is associated with a street segment first based on a distance measurement, an easier solution is possible. With the help of the parent-child relationship between community districts/police precincts and street segments in the DCP LION dataset, spatial aggregation based on the segment identifiers is much less computationally intensive than through point-in-polygon tests.

While many existing studies rely on the underlying spatial databases to compute spatial relationships on-the-fly (e.g., point-in-polygon test and nearest neighbor finding), these computations are very expensive and should be accelerated to improve the performance of online OLAP queries. Towards this end, we have developed several GPU based modules to efficiently compute the spatial relationships between the taxi trip pickup/drop-off locations and different types of polygons based on different criteria. The implementations details of the point-in-polygon test and nearest neighbor based spatial joins for the associations are described in detail in [21] and [22], respectively. Following the same framework, we can associate the point locations with street segments in the NYC DCP LION dataset based on the nearest neighbor principle. Actually we can pretty much re-use the nearest neighbor based association between points and polygons [22]. This can be further explained by using Fig. 2 where three types of spatial associations are illustrated. To assign a polygon identifier to a point location based on the nearest neighbor principle (Fig. 2C), all the polygons that intersect with the window centered at the point is first identified. For each of the intersecting polygon, the distance between the point and the polygon is computed and the polygon that has the shortest distance with the point is associated. To calculate the distance between a point and a polygon, the shortest distance between the point and all the edges of the polygons is used. The same procedure can be used to associate points with street segments as illustrated in Fig. 2A as both polygon boundaries and street segments can be considered as polylines. Since the majority of the street segments have only one edge while a polygon has at least three edges, the computation overhead is smaller. The approach to associate points with polylines can also be used for point-in-polygon test as illustrated in Fig. 2B ([21] provides more details). From now on, we assume that each point location is associated with an identifier for each type of infrastructure data, such as street segments in Fig. 2A, census blocks in Fig. 2B and tax blocks in Fig. 2C.

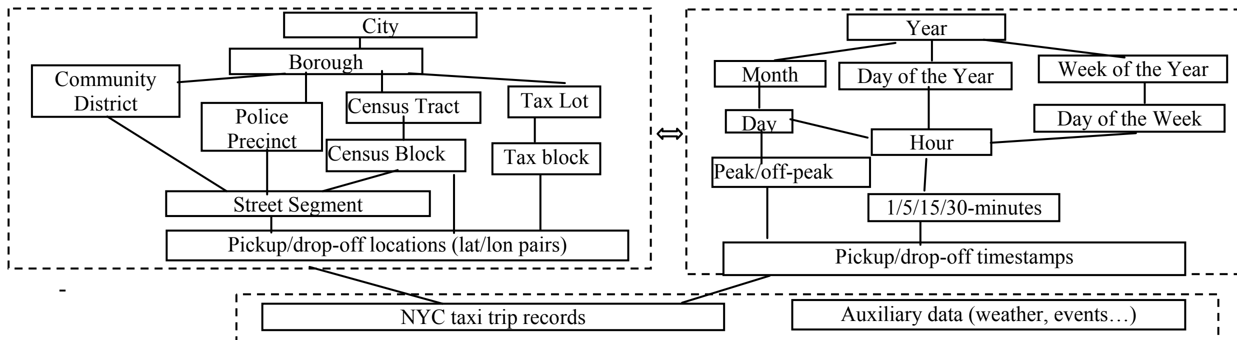
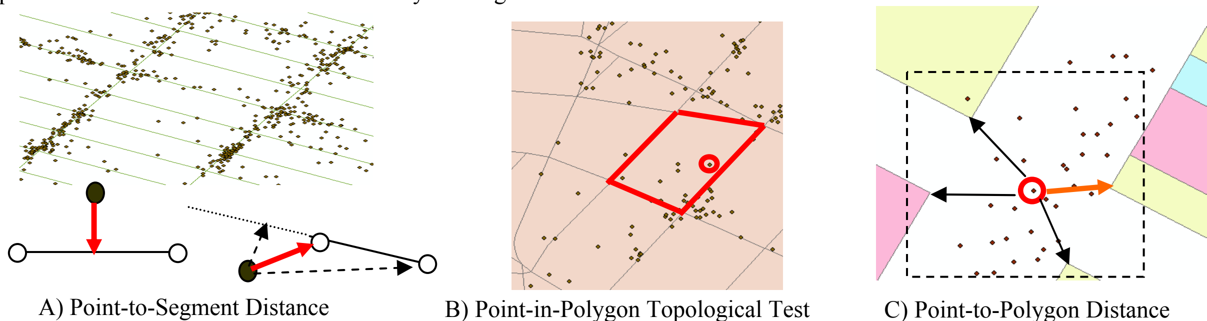


Fig. 1 Illustration of Spatial and Temporal Aggregation Hierarchies

Clearly there is a tradeoff between materializing higher levels of identifiers (e.g., census tracts and tax lots) and looking them up on-the-fly based on the semantic hierarchical relationships (e.g., parent→child relationships in Census Tract→Census Block and Tax Lot→Tax Block). Our design is to materialize the high-level identifiers and store them on disks in the same way as the bottom level identifiers but it is up to the query optimizer to decide whether to use them by loading them from disks to CPU/GPU main memories. When there are sufficient memory capacities left, it may still be beneficial to load the materialized high-level identifiers from disks. Given that many high-level identifiers are duplicated, generic data compression techniques can be applied to further reduce disk storage and I/O overheads and we refer to [31][32] for CPU and GPU based compressions on relational data in a database setting.



A) Point-to-Segment Distance B) Point-in-Polygon Topological Test C) Point-to-Polygon Distance

Fig.2 Illustration of Three Types of Point to Infrastructure Data Association for Spatial Aggregations

3.2 Compression and Temporal Aggregations

After introducing the design considerations of spatial aggregations, we now turn to temporal aggregations. Although the temporal hierarchy shown in Fig. 1 is more universal and easier to understand than the spatial hierarchy we have used in this study, given that time functions are not supported on GPUs, a technical challenge is to develop a compact time representation that has small memory footprint and is GPU computing friendly when it comes to temporal aggregation. We note that 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 the 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 can not be used on GPUs directly which brought a signficant 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 bit shifting which is 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 the week" and "day of the year" very well as these two fields are not explicitly stored as in "*struct tm*" in the standard C language. Computing the values of these two fields, while feasible on CPUs (by using C/C++ *mktime* function), can incur signficant overheads when the number of records is huge. For example, our experiments have shown that computing "day of the 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 the week" or "day of the year". Since the possible combinations of (year, month, day) in a reasonably long period is limited (in the orders of a few thousands), they can be aggregated to "day of the week" or "day of the 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 the week" or "day of the year" from "year+month+day" which is nontrivial.

The distinctions between "peak" and "off-peak" are often combined with the distinctions among week days and weekends as well as different types of holidays which make temporal hierarchies a little more complex in this case. However, the possible number of combinations does not depend on the number of data records and can be hardcoded into lookup tables to keep track of the child-parent relationships (m:1) when necessary on both CPUs and GPUs. In particular, these read-only, small sized lookup tables can be stored in special read-only texture memories to speed up the lookups on Nvidia GPUs.

4 IMPLEMENTATION DETAILS

4.1 Overview

It is beyond our scope to implement all the types of the spatial, temporal and spatiotemporal aggregations that have been modeled in the literature [5][33]. Instead, we focus on the implementations of aggregations along the spatial and temporal hierarchies shown in Fig. 1 and discussed in Section 3. We divide an aggregation into two phases, i.e., the association phase and the counting 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 fare and tip), offline materialization becomes infeasible and fast real-time online aggregations become critical. In addition, when the 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 associations are more desirable in such cases.

In this study, as discussed in Section 3.1, we will re-use the framework of GPU-based spatial join to associate taxi pickup and drop-off locations with their nearest street segments or polygons for further aggregations along the spatial and temporal hierarchies. Due to space limit, we will not repeat the implementation of the GPU based spatial join algorithms but we refer to [21] and [22] for details. We will report the experiment results on spatially joining pickup locations with street segments in Section 5.1 as the experiments have not been done previously. Before we introduce the implementation details of the counting phase on both multi-core CPUs and many-core GPUs in the next two subsections, we would like to note that the number of bins involved in the aggregations listed in Fig. 1 range from a few hundreds of thousands (e.g., tax block and street segments) to a few (e.g., days of the week) and some are fixed (e.g., 24 hours in a day) while some others are variable (e.g., street segments) depending on data semantics.

4.2 Parallel Counting on Multi-Core CPUs

Although it is more convenient to use Standard Template Library (STL) [34] map data structure to store (key, count) pairs for counting, we have found that using simple linear data structures such as arrays is much more efficient. While using arrays requires knowing array lengths beforehand, typically this is not a problem for OLAP applications as the metadata information of each dimension is often known (or the upper bound can be estimated). For comparison purposes, we have implemented a same aggregation using both the STL map data structure and arrays on CPUs as reported in Section 5. To parallelize the counting phase on CPUs, we have used OpenMP directives (e.g., *pragma omp for*). To avoid write conflicts, each thread is given its own private variables (arrays or STL maps) before the individual aggregated results are combined into the final results. Note that this is only possible when the number of threads is small (i.e., on CPUs). The approach to avoid write conflicts is not scalable in general and can not be applied to GPUs where hundreds of thousands of threads may be launched simultaneously. Since a

dual quadcore machine has 8 cores and each core supports 2 software threads using the Intel hyper-threading technology, we have varied the numbers of threads from 1, 2, 4, 8 to 16 for comparison purposes. Using 16 threads, which is twice the number of cores, may or may not improve overall performance depending on the degree of resource contentions.

4.3 Parallel Counting on Many-Core GPUs

Two options are available to implement multidimensional aggregations on GPUs, one is based on native parallel programming languages such as CUDA [35] and one is using parallel primitives that are developed on top of the native parallel programming languages. The first option, which requires a deep understanding of GPU hardware details and high parallel programming skills, has been adopted in [1]. 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 native programming languages as reported in [1]. Another research effort on MOLAP cube [27] is closely related to OLAP processing. The research utilizes parallel primitives that have been implemented in the CUDPP parallel library [36] which has allowed the authors to focus on high-level constructs without diving into too many hardware details. Based on these two pioneering works, we have decided to adopt a parallel primitive based approach by using the Thrust parallel library [37] as much as possible for fast prototyping and future portability. We note that Thrust is now part of CUDA SDK and the functionality and efficiency have been significantly enriched in its recent releases. That being said, we are aware that there are tradeoffs between code efficiency and coding complexity of parallel primitives based implementations versus native CUDA based implementations and we plan to compare the performance differences in our future work. We next introduce the parallel primitive based GPU implementation using the Thrust library in more details.

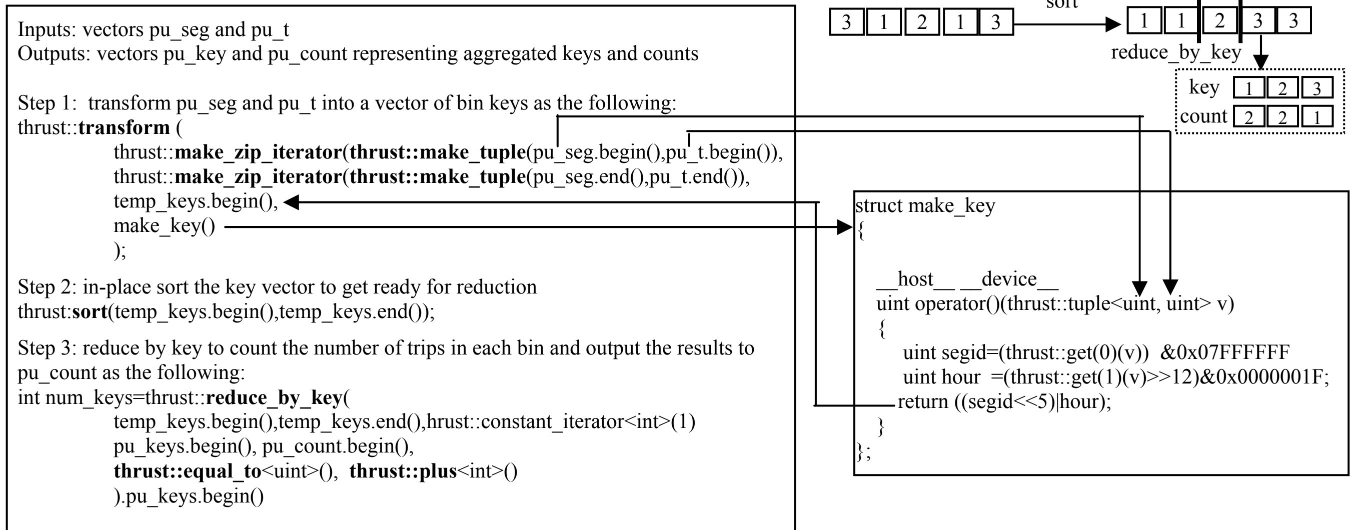


Fig. 3 Algorithm Design and Code Segment of a Spatiotemporal Aggregation using Parallel Primitives

Since 1-dimensional arrays work best for the current generations of Nvidia GPUs, the data arrays on CPUs that have been used for aggregations are copied to GPUs. The aggregations can be nicely mapped to the *reduce_by_key* parallel primitive that is supported by Thrust. Together with some other parallel primitives (such as *transform* and *sort*), iterators (such as *constant*, and *zip*) and user defined functors (C++ function objects), these aggregations typically can be implemented in a few lines. The Thrust library handles all thread organization, kernel launching and performance optimization issues transparently to developers. While it is beyond the scope of this paper to present the details of primitives based parallel programming, we refer to the Thrust website [36] for more details. The appendix of [21] also provides a brief introduction to several parallel primitives that are involved in the GPU implementation of the aggregations and can be a source of reference if interested.

Assume 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. We are given two vectors with the first storing the street segments (after spatial association) and the second storing the pickup time in the compressed form (Section 3.2) for all taxi trip records using a column-oriented data layout. The task can be completed using a few parallel primitives as illustrated in the algorithm design and code segment given in Fig. 3. 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 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 identifier 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. 3. After the 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 *hrust::constant_iterator*). The primitive allows user 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 *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 to make keys in most cases. Experiments have shown that the sorting time usually dominates the whole aggregation processes and we would like to discuss the suitability of using GPU for sorting. For this purpose, we refer to [38] for benchmarking results on GPU-based sorting for 32/64 bit keys. The results have shown that more than 1 billion per second sorting rate has been achieved on Nvidia GTX 480 for 32 bit keys which is considerably faster than single core CPUs can achieve. 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 is accessed multiple times when different primitives are invoked. Nevertheless, we consider the tradeoff is well justified from several practical aspects.

5 EXPERIMENTS AND RESULTS

5.1 Data and experiment setup

Through a partnership with the New York City (NYC) Taxi and Limousine Commission (TLC), we have access to roughly 300 million GPS-based trip records 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 performance of associating the pickup locations with census block polygons and tax block polygons for spatial associations have been reported in [21] and [22], respectively. The GPU performance has been compared with CPU or hybrid implementations as well. In this study, the GPU-based implementation on associating the taxi pickup locations with 147,011 street segments to compute the spatial relationships between taxi pickup locations and street segments follows a very similar approach as discussed in Section 3.1. All experiments are performed on a Dell Precision T5400 workstation equipped with dual quadcore CPUs running at 2.26 GHZ with16 GB memory, a 500GB hard drive and an Nvidia Quadra 6000 GPU device. The sustainable disk I/O speed is about 100 megabytes per second while the theoretical data transfer speed between the CPU and the GPU devices is 4 gigabytes per second through a PCI-E card.

5.2 Results on Spatial Association on GPUs

Among the 168,379,168 taxi pickup locations in NYC, the majority are successfully associated with their nearest street segments within D=250 feet. However, there are 867,163 locations have computed shortest distances that are more than 250 feet which are considered as outliers (0.515%) and are excluded from subsequent analysis. With respect to runtime, similar to what have been discussed in detail in our previous works [21][22], there are three components involved, i.e., generating point quadrants (t1), filtering bounding boxes of both point quadrants and street segments (t2) and distance computation and identifier assignment (t3). Using the maximum point quadrant size K= 512, the runtimes for the three components are listed in Table 1 where the columns indicate the numbers of months in the year (2009) that are used in the spatial associations. We can see that t1 dominates the total runtime in all tests and increases almost linearly with the number of point locations. We note that t1 has already included data transfer times between CPUs and GPUs which count nearly 25% of the end-to-end runtime using 12 months data (the whole year of 2009). Since it takes about 15 seconds for 12 months and less than 5 seconds for 3 months, we conclude that spatial associations on GPUs can achieve near real-time responses and are suitable for online aggregations (the counting time is negligible as can be seen from the results in the next two subsections).

These results, together with our previous results in associating the 168.38 million point locations with 43,252 census block polygons (11.165 seconds) [21] and 735,488 tax block polygons (33.110 seconds) [22] suggest that the performance boosting techniques on modern hardware can have great potentials in speeding up processing of large-scale geo-referenced data. Given the performance at the yearly level (in the order of 10-40 seconds), we are positive that interactive spatial associations (spatial join queries) may be possible at the monthly level on low-end commodity GPUs with small memory capacities (e.g., 1 GB). This also makes it possible to scale-out to larger datasets by adopting distributed computing using a shared-nothing framework as discussed in Section 2 and we leave it for our future work.

Table 1 Results on Spatial Associations on GPUs

# of Months	1	2	3	4	6	9	12
N1 ($*10^6$)	13.84	27.00	41.17	55.23	83.81	124.64	168.38
N2 ($*10^6$)	0.155	0.306	0.496	0.676	0.982	1.358	1.747
t1 (second)	0.955	1.876	2.908	3.915	5.986	9.001	12.233
t2 (second)	2.059	1.615	1.472	1.495	1.123	1.176	1.221
t3(second)	0.200	0.343	0.519	0.677	0.941	1.270	1.601
T=t1+t2+t3	3.214	3.834	4.899	6.087	8.050	11.447	15.055

Note: N1- # of point locations; N2- # of point quadrants

5.3 Results on Parallel Counting on CPUs

We have performed six groups of aggregations on multi-core CPUs with three types of aggregations and two types of data structures (STL container class and array). The three types of aggregations are the following: counting on the 147,011 street segments using both STL and array (spatial aggregation), counting on the 24 hours (temporal aggregation) and counting on both street segments and hours (spatiotemporal aggregation). We note that the street segment identifiers in the DCP LION dataset are not numbered continuously due to regular quarterly updates. However, the largest number is 175,440 which is not far way from the number of street segments identifiers and we can use a dynamically allocated array for the counting purpose. The purposes of the experiments are the following (1) Is it feasible to perform real-time aggregations on CPUs? (2) What are the differences in using STL container classes that requires extensive dynamic memory allocations and simple linear data structures like arrays? (3) What is the scalability of using multiple threads on multi-core CPUs for the aggregations? Clearly, the number of bins in the spatial aggregation based on segment identifiers is 3-4 orders larger than the number of bins in the temporal aggregations based on hours and can be used to represent the two extremes in aggregations with respect to cardinality. The results are summarized in Table 2 where columns 1T/2T/4T/8T/16T refer to using 1, 2, 4, 8 and 16 threads, respectively.

From Table 2 we can see that, the array based serial implementations are about 23X, 13X and 29X faster than the STL based serial implementations for the three types of aggregations. The same trends can be observed when comparing parallel implementations using different numbers of threads although the speedups decrease as the numbers of threads increase. The results seem to suggest that the larger the bins, the greater the speedups for array based implementations over the STL based

implementations. This is not surprising in the sense that memory accesses are becoming increasingly expensive on modern processors when compared to computing [3]. Dynamic memory allocations and deallocations are not only costly but also result in "pointer-chasing" problem especially when memory footprints are large. This may in turn incur signficant cache misses due to irregular memory accesses. We thus, from a practical perspective, advocate simple linear data structures for fast in-memory scans and avoid complex data structures (e.g., indexing trees and aggregation trees) unless there are clear performance advantages.

From Table 2 we can also see that the performance of the six parallel aggregations increases sub-linearly with the number of threads on multi-core CPUs, especially when the number of threads is increased from 8 to 16 where performance can even drop. This is expected as threads, especially the two software threads within a single core, may compete for resources, including memory bandwidth. It can also be observed that the speedups (up to 7X) for counting using STL are generally higher than that of using array (about 2X). This might be due to the reason that the intensity of memory bandwidth competitions for the array based implementations are higher as the processing speed (and hence data movement along the memory bandwidth) 13-29 times higher. The memory bandwidth limit may be an important factor in fully utilizing the parallel processing power of multi-core CPUs. Given that counting on 170 million records only requires about 1/4 second for both spatial and temporal aggregations, which amounts to a 680 million per second rate on a dual quadcore commodity machine, we are positive that a single multi-core machine can achieve signficant throughputs in OLAP processing if the parallel processing power of SMP parallel hardware is fully utilized.

Table 2 Experiment Results for Different Aggregations on Multi-Core CPUs (in Seconds)

Implementation	Aggregation	Serial	1T	2T	4T	8T	16T
STL	1 Pickup Segment (spatial)	12.519	19.776	9.768	4.992	2.513	1.721
	2 Pickup Hour (temporal)	7.043	6.089	4.347	2.121	1.186	0.907
	3 Pickup Segment+Hour (Spatiotemporal)	17.128	24.238	12.522	6.707	3.803	3.781
ARRAY	4 Pickup Segment (spatial)	0.550	0.548	0.228	0.212	0.206	0.215
	5 Pickup Hour (temporal)	0.524	0.511	0.644	0.477	0.356	0.258
	6Pickup Segment+Hour (Spatiotemporal)	0.582	0.587	0.322	0.279	0.324	0.446

5.4 Results on Parallel Counting on GPUs

We have performed the same spatial, temporal and spatiotemporal aggregations on GPUs. The results show that the hourly temporal aggregation requires 0.257 second which is 2X faster than the serial array based CPU implementation but is comparable to the best parallel CPU implementation using 4/16 threads. The spatial aggregation requires 0.188 second which is about 3X times faster than the array based serial implementation but is only marginally better than the best parallel CPU implementation using 16 threads. For the spatiotemporal aggregation, the GPU runtime is 0.274 second which is 2.14X faster than the serial CPU implementation but again it is comparable with the best parallel CPU implementation using 4 threads. While the comparisons do not suggest that many-core GPU is a clear winner over multi-core CPU as previous research has suggested, we have to bear in mind that our aggregation implementations are based on high-level parallel primitives in a way similar to STL classes. If we compare the respective runtimes for GPU and STL implementations, the GPU implementations still gain 9.2X, 3.5X and 13.8X for spatial, temporal and spatiotemporal aggregations.

Although the parallel primitives based implementations on many-core GPUs are only reasonably better than multi-core CPUs in the counting phase, our implementation of the spatial associations have achieved 3-4 orders better performance on many-core GPUs than the implementation using an existing database. Since the runtimes of the counting phase are fairly insignificant when compared with that of the association phase (1/4 second v.s. 15 seconds), the overall speedups are still signficant with respect to the end-to-end performance. On the other hand, if only the runtimes in the counting phase are included in the query times, which is typical in the cases where spatial relationships are materialized in a static setting as discussed in Section 5.1, since the performance of many-core GPUs and multi-core CPUs are comparable, integrating the two types of SMP processors to further improve the overall performance is desirable. However, the approach can be technically challenging and is left for our future work.

6 CONCLUSION AND FUTURE WORK

In this study, we report our designs, implementations and experiments on spatial, temporal and spatiotemporal aggregations of hundreds of millions of taxi trip records in an OLAP setting. By utilizing the massively data parallel GPU processing power, we were able to spatially associate nearly 170 million taxi pickup location points with their nearest street segments among 147,011 candidates in about 15 seconds. Spatial, temporal and spatiotemporal aggregations can be processed in a fraction of a second on both multi-core CPUs and many-core 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 by intelligently integrate the two types of SMP processors with distinct hardware features.

For future work, first of all, to scale up, we would like to further reduce the processing times for both spatial association and counting. Second, to ensure usability, we would like to adopt an engineering approach to investigate the appropriate spatial and temporal scales so that interactive OLAP processing can be smoothly performed on commodity personal computers with different hardware configurations. Finally, to scale-out, we plan to explore cluster computing technologies to process larger scale data, for example, multi-year and multi-city.

References

1. Tobias, L., Amitava, D., Zurab, K. and Christoffer, A. (2010). Exploring graphics processing units as parallel coprocessors for online aggregation. Proceedings of ACM DOLAP Workshop.
2. Wrembel, R., Aufaure, M.-A. and Zimányi, E. (2012). Data Warehouse Performance: Selected Techniques and Data Structures. Proceedings of eBISS 2011 (Springer LNBIP 96), 27-62.

3. Hennessy, J.L. and Patterson, D. A, 2011. Computer Architecture: A Quantitative Approach (5th ed.). Morgan Kaufmann.
4. Metropolitan Transportation Authority (2012). Subway and Bus Ridership. http://www.mta.info/nyct/facts/ridership/index.htm
5. Vaisman, A. and Zimnyi, E. (2009). What Is Spatio-Temporal Data Warehousing? Proceedings of DaWak Conference.
6. Rivest S., Bédard Y., Marchand P., 2001, Toward Better Support For Spatial Decision Making: Defining the Characteristics of Spatial On-Line Analytical Processing (SOLAP), Geomatica, 55(4), 539 -555
7. Alberto, O. M. and Alejandro, A. V. (2000). Temporal Queries in OLAP. Proceedings of VLDB Conference.
8. Fidalgo, R., Times, et al(2004). GeoDWFrame: A Framework for Guiding the Design of Geographical Dimensional Schemas. Proceedings of DaWak Conference.
9. Kamal, B., Sandro, B., Hadj, M. and Francois, P. (2010). Towards the definition of spatial data warehouses integrity constraints with spatial OCL. Proceedings of ACM DOLAP workshop.
10. Gomez, L., Haesevoets, S., Kuijpers, B. and Vaisman, A. A. (2009). Spatial aggregation: Data model and implementation. Information Systems 34(6): 551-576.
11. Ariel, E., Leticia, G., Bart, K. and Alejandro, A. V. (2007). Piet: a GIS-OLAP implementation. Proceedings of ACM DOLAP.
12. Bimonte, S., Tchounikine, A. and Miquel, M. (2007). Spatial OLAP: Open Issues and a Web Based Prototype. Proceedings of the AGILE Conference on GIS.
13. Scotch, M. and Parmanto, B. (2005). SOVAT: Spatial OLAP Visualization and Analysis Tool. Proceedings of the Hawaii International Conference on System Sciences.
14. Octavio, G., Jose-Norberto, et al (2010). Using web-based personalization on spatial data warehouses. Proceedings of the EDBT/ICDT Workshops
15. Siqueira, T. L., Ciferri et al (2012). The SB-index and the HSB-index: efficient indices for spatial data warehouses. Geoinformatica 16(1): 165-205.
16. Brito, J. J., Siqueira, T. L. L., et al (2011). Efficient processing of drill-across queries over geographic data warehouses. Proceedings of DaWaK Conference.
17. Kenneth, C. and Wo-Shun, L. (2008). Processing Aggregate Queries on Spatial OLAP Data. Proceedings of DaWaK.
18. Dimitris, P., Panos, K., Jun, Z. and Yufei, T. (2001). Efficient OLAP Operations in Spatial Data Warehouses. Proceedings of SSTD Conference.
19. http://postgis.refractions.net/
20. Zhang, J., Gong, H. et al (2012). U2SOD-DB: A Database System to Manage Large-Scale Ubiquitous Urban Sensing Origin-Destination Data. To appear in the Proceedings of ACM SIGKDD Workshop on Urban Computing.
21. Zhang, J. and You., S. (2012). Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. Technical report online at http://geoteci.engr.ccny.cuny.edu/pub/pipsp_tr.pdf
22. Zhang, J, You, S. and Gruenwald, L. (2012). High-Performance Spatial Join Processing on GPGPUs with Applications to Large-Scale Taxi Trip Data. Technical report online at http://geoteci.engr.ccny.cuny.edu/pub/nnsp_tr.pdf
23. http://en.wikipedia.org/wiki/Intel_MIC
24. http://en.wikipedia.org/wiki/GeForce_600_Series
25. Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D. and Moon, B. (2012). Parallel data processing with MapReduce: a survey. SIGMOD Record, 40 (4), 11-20.
26. Grund, M., Kruger, J., Plattner, H.,et al (2010). HYRISE: a main memory hybrid storage engine.Proceedings of the VLDB Endowment 4(2): 105-116.
27. Krzyszto, K. and Rudny, T. (2011). MOLAP cube based on parallel scan algorithm. Proceedings of ADBIS.
28. Kaczmarski, K. (2011). Comparing GPU and CPU in OLAP cubes creation. Proceedings of SOFSEM.
29. Sitaridi, E. A. and Ross, K. A. (2012). Ameliorating memory contention of OLAP operators on GPU processors. Proceedings of DaMoN.
30. http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml
31. Holloway, A. L., Raman, V., et al. 2007. How to Barter Bits for Chronons: Compression and Bandwidth Tradeoffs for Database Scans. Proceedings ACM SIDMOD Conference.
32. Fang, W., He, B. and Luo, Q. (2010). Database compression on graphics processors. Proceedings of VLDB Endowment.3(1-2): 670-680
33. Ines Fernando Vega, L., Richard, T. S. and Bongki, M. (2005). Spatiotemporal Aggregate Computation: A Survey. IEEE TKDE, 17(2): 271-286.
34. http://www.sgi.com/tech/stl/
35. http://www.nvidia.com/object/cuda_home_new.html
36. http://code.google.com/p/cudpp/
37. http://thrust.github.com/
38. Merrill, D. and Grimshaw, A. (2011). High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters 21(2): 245-272. Also online at http://code.google.com/p/back40computing/wiki/RadixSorting

Appendix SQL Statements for spatial/temporal aggregations in PostgreSQL (Q11 and Q15 are used as examples in the experiments)

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, lion09c
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, lion09c
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