U²STRA: High-Performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs

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

Volumes of GPS recorded trajectory data in ubiquitous urban sensing applications are increasing fast. Many trajectory queries are both I/O and computing intensive. In this study, we propose to develop the U²STRA prototype system to efficiently manage large-scale GPS trajectory data using General Purpose computing on Graphics Processing Units (GPGPU) technologies. Towards this end, we have developed a trajectory data layout schema using simple in-memory array structures which is not only flexible for data accesses but also cache friendly. We have further developed an end-to-end trajectory similarity query processing technique on GPUs. Our experiments on two publically available large trajectory datasets (GeoLife and T-Drive) have demonstrated the efficiency of massively data parallel GPGPU computing. An impressive 87X speedup for spatial aggregations of GPS point locations and 25-40X speedups for trajectory queries over serial CPU implementations have been achieved. The U²STRA system has also been integrated with commercial desktop and Web-based GIS systems and spatial databases for visual exploration purposes.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database applications – Data mining, Spatial databases and GIS

General Terms

Management, Design

Keywords

Ubiquitous Sensing, GPS Trajectory, High-Performance, GPGPU, Similarity Query, Spatial Aggregation

1. INTRODUCTION

Huge amounts of pervasive urban sensing data are being captured at ever growing rates due to the increasing availability of imaging, locating and other types of sensing technologies on portable wireless devices and increasing urban activities. In particular, Global Positioning System (GPS) traces have been recorded routinely by taxicabs in many big cities over the world. For example, The T-Drive sample dataset collected by Microsoft Research Asia [1] has 15 million GPS readings from 10,357 taxis during a single a week and the dataset amounts to 762 MB in text format. The number of GPS point locations compiled by OpenStreetMap from GPS traces contributed by world-wide volunteers in the first seven and a half years has reached 2.77 billion and the data volume is 55 GB [2]. Nokia probe vehicles collects more than 11 billion GPS readings in major cities over the world to measure and predict traffic flows (Personal communication). Yet cell phone call logs represent a category of data at an even larger scale [3, 4]. These Ubiquitous Urban Sensing (U²S) data, while very useful in understanding a variety of aspects of urban dynamics, traffic conditions and social interactions, have imposed signficant challenges on data management.

Geographical Information System (GIS) and Spatial Databases (SDB) are commonly used techniques in managing geo-referenced data. Unfortunately, most of the existing commercial and open source GIS and spatial databases are disk-resident and are designed for managing transactional data. The performance is usually rather poor in managing large-scale trajectory data. In recent years, quite a few Moving Object Databases (MOD) techniques have been developed to index and query trajectory data [5, 6, 7, 8, 9]. However, most of them are designed based on traditional database architectures, i.e., disk-resident and serial CPU computing.

To achieve the desired level of high-performance in querying and data mining of large-scale trajectory data, it is natural to explore the massively data parallel General Purpose computing on Graphics Processing Units (GPGPU) technologies. Despite the fact that almost all reasonably current desktop and server computers have already been equipped with GPU devices that are capable of general computing and there have been many successful applications in different domains, there is relatively little research on using GPGPU technologies for data management. Among the few pioneering works to explore the potentials of using GPGPU computing power for data management, the majority focuses on indexing and query processing on relational data [10, 11]. The work reported in this paper is an extension to our previous work on using GPGPU technologies for managing U²S Origin-Destination (OD) data [12]. Our experiments have demonstrated 3-4 orders speedups when joining point locations with urban infrastructure data (street networks and different type of zones) based on different criteria [13,14, 15]. In this study, we extend our experiences in designing and developing the U²SOD-DB system to trajectory data in the same context of managing ubiquitous urban sensing data (where urban infrastructure plays a key role) and we call our prototype system as U²STRA.

The rest of the paper is arranged as the following. Section 2 introduces background and related work. Section 3 presents the system architecture and discusses some design considerations. Section 4 provides the technical details of the new implementations of key components. Section 5 provides some experiment results. Finally Section 6 is conclusion and future work.

CDMW'12, October 29, 2012, Maui, Hawaii, USA.

2. BACKGROUND, MOTIVATIONS AND RELATED WORK

Moving objects and trajectories have attracted considerable research interests in the past decade [5, 6, 7, 8]. While moving objects are not necessarily constrained by infrastructure (such as road networks), trajectories recorded by GPS devices, cellular networks or wifi networks, which are typical in the context of ubiquitous sensing and computing, are strongly connected with urban infrastructure data, including street segments, Points of Interests (POIs) and land use types. In general, in contrast to other types of trajectories, such as hurricane paths and animal movements, U^2S trajectories are closely related to human activities with specific trip purposes. While the rich semantics make U^2S trajectory data interesting in various applications, significantly more sophisticated data management techniques are required to make sense out of the huge amount of the data due to the sophistications of human activities [16, 17,18].

Spatial, temporal and spatiotemporal query processing is fundamental to trajectory data management and analysis. Quite a few indexing techniques have been developed to speed up processing the queries over the past few years and we refer to [5, 6, 7, 8, 9] for reviews. More specifically, SECONDO is an extensible database system that has been extensively used to manage moving objects and trajectories [19, 20]. Recently, the M-Atlas project has made its system available for download [21] which is built on top of the open source PostgreSQL database [22] and its PostGIS plugin [23]. These indexing techniques and system realizations target at different types of queries. In this study, we are particular interested in similarity related trajectory queries which have also received extensive attentions [24-27]. We currently use Hausdorff distance whose behaviors have been discussed in previous studies (e.g. [28]) and we are also actively exploring various definitions of trajectory similarities.

The majority of the existing indexing structures and query processing algorithms are designed for serial CPU implementations. The designs usually favor using sophisticated algorithms in reducing computation overheads and improving query response times. This is a natural choice before multi-core CPUs and many-core GPUs become the mainstream commodity processors. Unfortunately, there are situations that efficient but sophisticated data structures and algorithms may perform poorly on parallel hardware. Parallelization on such query processing algorithms can be very difficult if excessive coordination is required to utilize parallel processing units. Furthermore, due to the increasing gap between computing and I/Os [29], simple data structures such as arrays and linear scanning may outperform indexing that require non-sequential data accesses in certain cases due to the nature of caching mechanisms on modern hardware architectures. Another new technical trend on modern hardware is the increasingly availability of large memories which makes it possible to quickly stream large chunks of data between memory and disks. This may largely reduce the need for page-based buffer management in traditional disk-resident databases, especially in an Online Analytical Processing (OLAP [30]) setting.

Based on these observations, our goal is to design a prototype system that can utilize commodity hardware capacities, including parallel computing power and large memory capacity, to boost the performance of OLAP type queries in a batch mode for U^2S trajectory data. Instead of limiting to multi-core CPUs, we have chosen to use GPUs as co-processor for more computationally intensive modules, such as distance based joins. We note that a distance computation may require significantly fewer clock cycles on GPUs than on CPUs due to their special hardware designs. In addition, GPUs usually have signficant larger numbers of processing cores than CPUs. For example, the Nvidia GeForce GTX 690 GPUs [31] that are currently available form the market have 3072 cores. Although GPU processors (~ 1 GHZ) are typically weaker than CPU processors, thousands of processors together can deliver huge amounts of computing power than CPUs and even small cluster computers. The combined fast floating point computing power and large number of processors make GPUs suitable for accelerating trajectory queries that involve large amount of distance computation.

While nearest neighbor searching have been extensively used on GPUs for various applications [33-37], it seems that there are few previous works on speeding up spatial and spatiotemporal queries that require large amounts of distance computation on GPUs. The potential of GPU accelerations in speeding up queries on trajectory data in a database environment (by utilizing indices for filtering) is largely unknown. We believe our prototype system can provide a concrete case study on this aspect. Our proposed research and implementation can be used to evaluate the relative advantages and disadvantages of classic efficiency oriented design and the new design based on the throughput oriented GPU computing paradigm [38] in the context of managing large-scale trajectory data.

3. SYSTEM ARCHITECTURE DESGIN

In addition to designing an implementable architecture to handle unique characteristics of GPS-based trajectory data as reported next, we have brought our previous experiences in designing the U^2SOD-DB for origin-destination data [12] into the U^2STRA system for trajectory data. These experiences include timestamp compression [12, 14], array-based simple in-memory structures and parallel primitive friendly design for fast implementation [13, 15]. We also note that our current design on trajectory data is based on our experiences in processing the T-Drive [1] and the GeoLife (also from Microsoft Research Asia [39]) GPS trajectory datasets. We are working on further abstracting the design to accommodate for more general cases.

The overall system design is illustrated in Fig. 1. Before we present the design of the key components, we would like to introduce the array-based trajectory representation which is fundamental to the system design. While a widely accepted trajectory representation is still lacking, following the Open Geospatial Consortium (OGC) Simple Feature Specification (SFS) [40] on polygons, we have defined the following four-level hierarchy to represent trajectory data, i.e., dataset→trajectory→track→point. A trajectory dataset is a collection of trajectories and a trajectory is a collection of tracks where each track comprises a sequence of points. The criteria on the divisions among the first three levels can be flexile which largely depend on applications. A point has at least three attributes (x,y and t) but allows additional attributes. Similar to using simple arrays to represent polygons (whose benefits are discussed in [14]), we use the following four arrays to represent the four-level hierarchy. First of all, by accessing the Trajectory Index (TRI) array, we know the starting and ending positions (and hence the number of trajectories) of the i^{th} dataset. For

example, in Fig. 2, the 12th dataset (base 0) has 10 trajectories that begin at the 50th trajectory (inclusive) and ends at the 60th trajectory (exclusive). Similarly, the Track Index (TKI) array stores the starting positions all trajectories and the Point Index (PTI) array stores the starting positions of all tracks. By accessing the 50th elements in TKI, we know that the 50th trajectory has 27 tracks with a starting position of 73. Correspondingly, the 73rd track has 76 points with a starting position at 913 in the point array. By accessing the point array, we can retrieve the respective values of x/y/t and other attributes. We note the point array can be implemented as Array of Structures (AoS) or Structure of Arrays (SoA) depending on how often the x/y/t components (and other relevant components if present) are used together. The design also makes it easy to associate additional attributes at the dataset, trajectory and track levels by providing additional arrays with each element correspond to the indices in the TRI, TKI and PTI, respectively. For example, in the Microsoft Research Asia GeoLife dataset [38], some trajectories are manually labeled with travel modes which are very useful for analysis. These travel mode labels can be put into an array that corresponds to the trajectory or track index array (TRI and TKI, respectively) so that the travel mode of each trajectory/track can be easily retrieved by simply accessing the arrays using a position index. We note that since we only keep the beginning positions in TRI/TKI/PTI and we rely on the next positions to compute the lengths of the corresponding datasets (for numbers of trajectories), trajectories (for numbers of tracks) and tracks (for numbers of points), they need to be put in the respective array in an ordered manner to establish the correspondences. On the other hand, if the index arrays are extended to include both the starting and the ending positions (or lengths), it becomes possible to build subsets of the trajectory data by providing multiple sets of TRI/TKI/PTI arrays but reusing the point array. This can be convenient and efficient in some application scenarios. Of course, it is always possible to extract partial of the trajectory data, rebuild the four arrays and use them as a completely new trajectory store.

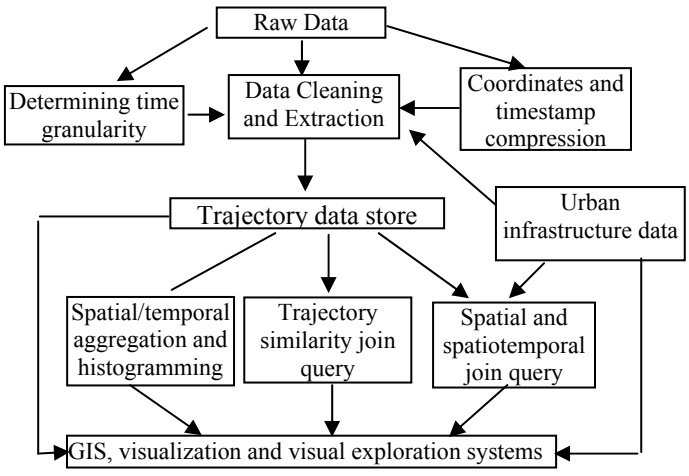

Fig. 1 Overview of U²STRA System Architecture

After transforming raw data into a structured trajectory store, a variety of queries can be performed. In the current implementation of U²STRA, we have limited ourselves to three types of queries as shown in the middle part of Fig. 1. First, some simple yet useful statistics can be derived from the trajectory data through spatial queries (e.g. Minimum Bounding Boxes or MBRs for tracks/trajectories), temporal queries (e.g., durations for tracks/trajectories) or their combinations. Furthermore, by treating points on trajectories individually, we can aggregate these points spatially using different levels of grids in a way similar to using the point locations at the origins and destinations of taxi trips in U²SOD-DB [12]. These spatially aggregated grids can be filtered by different temporal units to generate daily or hourly grids to understand the overall patterns of the GPS trajectory data.

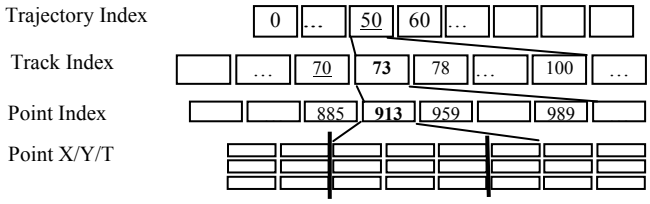

Fig. 2 Array Representation of Trajectory Data

Second, the trajectories can be joined with urban infrastructure data, such as POIs, road networks, administrative regions and census blocks based on different spatial and spatiotemporal relationships. Depending on applications, there can be many join criteria. For example, to count the number of trajectories (or tracks) that are completely within a region during a certain time period, we would require all the points in the trajectory/track to be in the region during the period. This can be realized by extending our previous design on point-in-polygon test [13] by applying an *AND* operator over the test results of all points on the trajectory/track. We are in the process of evaluating the possibilities of implementing more complex spatiotemporal queries (e.g., those discussed in [8] in a serial CPU computing based database setting) on GPUs based on our existing codebase (with necessary extensions) before we decide to include them in U²STRA.

In addition to adapting our GPGPU based spatial join to trajectories as we just discussed, our current major design and development efforts focus on similarity based trajectory join processing. While quite a few approaches to computing the similarity between two trajectories (and tracks) been defined, as mentioned earlier, we currently use Hausdorff distance which is defined as the maximum of minimum distances between two point sets as shown in the left part of Fig. 3. Among the minimum distances between the four points in T1 to T2, d3 is the largest one and will be used as the distance between T1 and T2. Following the filtering-refinement schema that has been extensively used in spatial databases [40], our idea is to use trajectory as the basic units for filtering based on the spatial relationships between their Minimum Bounding Boxes (MBRs). For two trajectories T1 and T2, as shown in the right part of Fig. 3, if the expanded MBR of a track (S_{1i}) in T1 (using an expansion distance D) and the MBR of a track (S_{2j}) in T2 overlaps then we further perform pair-wised distance computation and find the shortest distance to S_{2j} for all points in S_{1i}. Here the shortest distance between a point and a trajectory is simply defined as the minimum distance between the point and all points in the trajectory. After pair-wise distance computing is finished, each of the points in a trajectory (say T1) will have a shortest distance to one or more tracks in T2 (i.e., S_{2j}). By finding the maximum distance among the computed shortest

distances (i.e., maximum of minimum values), the distance between the two trajectories can be computed.

While the details of the implementation will be provided in Section 4, we would like to briefly discuss several design considerations. First of all, as we plan to implement the design on parallel hardware (GPUs to be more specific), the proposed design needs to be parallelization friendly. Pair-wise distance computation in the refinement phase is embarrassingly parallelizable. We also reuse the grid-file based spatial indexing for pairing trajectory tracks in the filtering phase by transforming a spatial relationship testing (intersecting) problem into a set of binary searching problems through equality test that is well supported in most parallel hardware including GPUs. Second, as can be seen in Fig. 2, the spatial join is performed at the track level which is in the middle between the trajectory and point levels. The design helps to effectively use the filtering power of MBRs.

Properly controlling the sizes of tracks in the preprocessing phase is very important. If the MBRs are too large, then the filtering power is limited which will result in quadratically growing numbers of pair-wise distance computation in the refinement phase. On the other hand, when the MBRs are too small, for a large query distance D, the expansion ratios will be large and will result in a large number of duplicated track pairs which need to be removed before the refinement phase. The large number of duplicated track pairs can impose signficant memory pressure on GPUs.

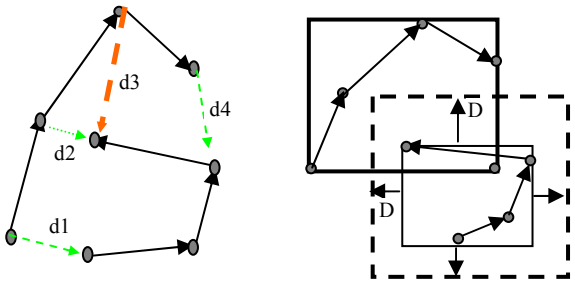

Fig. 3 Illustration of Hausdorff Distance Between Two Trajectories and Paring Two Trajectory Tracks

4. IMPLMENTAITON DETAILS

In this section, we will be focusing on the parallel implementations of trajectory similarity query on GPGPUs by assuming basic knowledge of GPGPU programming. We also refer to our related works for the implementation details on spatial and temporal aggregations [12, 14] and different types of spatial joins between points and urban infrastructure data [13, 14, 15]. Although the implementation of trajectory similarity query still follows the filtering-refinement schema in spatial joins [41], there are several unique features. First of all, trajectory similarity query involves a new data type (trajectory) and joins a trajectory dataset with itself (i.e., self join). Second, the similarity (using Hausdorff distance) is defined between two sets instead of between two individual objects as we have dealt in the previous studies. More importantly, unlike in point datasets where the MBRs of the divisions of the dataset do not overlap, the MBRs of trajectory tracks can overlap significantly. There will be multiple MBRs associated with a grid cell in the refinement phase (to be detailed shortly) which makes the implementation more complex.

The implementation of the trajectory similarity query begins with rasterizing the MBRs of trajectory tracks to a uniform grid. Based on the widths and heights of the MBRs, the numbers of rasterized grid cells for the MBRs can be determined (which are stored in a vector V1). In our GPU-based implementation, we have developed a GPU kernel (program block that can be executed in parallel) for this purpose by assigning a computing block to process a MBR. After applying an exclusive scan on V1, we can obtain the starting positions to output the cells of the MBRs (which are stored in V2). Each computing block then output the trajectory segment identifiers and rasterized cell identifiers based on V2, in parallel, to two vectors (VQQ and VQC), respectively. Please refer to the top-middle part of Fig. 3 for the illustration of this step. The second step is to rasterize the expanded MBRs (with the predefined distance D) of the trajectory segments by following the same procedure as in Step 1. The results are stored in VPP and VPC vectors, respectively, as illustrated in the bottom-middle part of Fig. 3. The third step is to pair the segment identifiers in VQQ and VPP through equality test on the cell identifiers in VQC and VPC. Our implementation is based on the vectroized binary search parallel primitive provided by the Thrust library [42] that comes with CUDA SDK as detailed below.

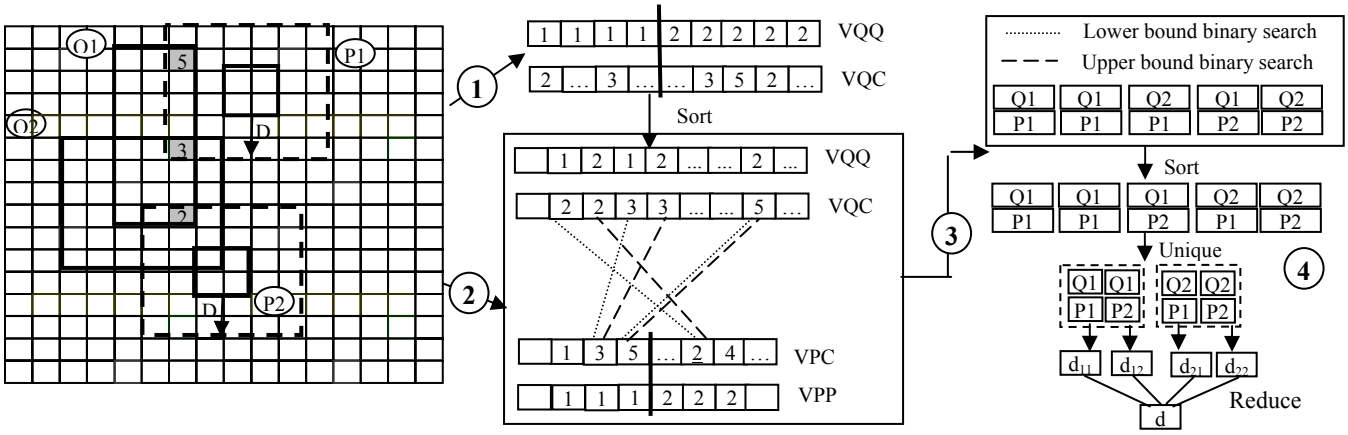

Fig. 4 Illustration of the Four Steps in GPU-based Implementation of the Filtering Phase for Trajectory Similarity Join Query

We assume VQQ are sorted based on VQC so that trajectory segments identifiers associated with a same cell identifier appear consecutively in VQC. A parallel sorting can be applied for this purpose. For each of the elements in VPC, in parallel, we perform a lower bound and an upper bound binary search on VQC. If there is a hit, then we know the starting and ending positions of the matched elements in VQQ. Based on these positions, the trajectory track identifiers stored in VQQ can be retrieved and they are paired with the trajectory track identifiers stored in VPP that corresponds to the element in VPC. As an example, assuming that we are searching the cell with an identifier of 2 in VPC (whose corresponding track identifier is 2 in VPP, i.e., P2), the lower bound and the upper bound of the binary searches are 1 and 2 respectively. The two segment identifiers stored in VQQ at the position range are 1 and 2 (i.e., Q1 and Q2), respectively. As such, P2 will be paired with Q1 and Q2. After knowing the number of matches for all elements in VPC, in a way similar to step 1, we can apply an exclusive scan to compute the output positions of the matched pair in the output vector (say V3) and copy the matched pairs to V3. It is conceivable that there will be duplicates in V3 with respect to the combinations of the track identifiers in Q and P. Since only one copy is needed to represent the matched track pair to be used in pair-wise distance computation in the refinement phase, in Step 4, we can follow a stand procedure by combining a sort and a unique primitive for this purpose.

After the filtering phase completes, we proceed with the refinement phase. We modify the pair-wise distance computing kernel we have developed previously [13-15] for this purpose. Within a CUDA computing block that handles a (P,Q) track pair, same as before, for each of the points in the Q (or P) track, we compute the minimum distance between the point and all the points in the P (or Q) track are computed. We assign a thread to each point in Q track and let it loop through all the point in the P track for this purpose. The maximum distance among all the S minimum distances is output where S is the number of points in the Q track. Finally, a maximum reduction on all the (P,Q) pairs to compute the global maximum distance among all the trajectory tracks of T1 and those of T2 if the bounding boxes are within the threshold distance D. Note that we have only shown two trajectories (P and Q respectively) in Fig. 4. In practice, there will be a large number of trajectories (and tracks) involved in a trajectory similarity query. This is not an issue since the track identifiers are globally unique and the parallel primitives do not limit the number of elements in the relevant vectors. To handle multiple trajectories, we will need to extend the last step slightly by looking up the trajectory identifiers based on the track identifiers and use a segmented version of the reduce primitive by using the combinations of the trajectory identifiers (from P tracks and Q tracks, respectively) as the key.

There are several technical issues need to be further discussed to better understand the filtering phase of the trajectory similarity join query processing. First of all, although we have used Q to represent tracks to be paired and P to represent tracks that initiates pairing and we have chosen to expand the MBRs of P elements, since this is a self-join and is symmetrical, it is possible to do the other way, i.e., expanding the MBRs of Q elements. Second, if we decide to expand the bounding boxes of P tracks, we note that there is no need to sort VPP based on VPC although a VPC element (cell identifier) may correspond to multiple track identifier as in the VQQ/VQC pairs. The reason is that each element in VPC searches through VQC independently (in parallel) and the paired result need to be sorted independently in Step 4 of the filtering phase. On the other hand, sorting on VQC/VQQ is necessary because the requirements of the binary search (including lower bound and upper bound search). Finally, we note that while the rasterization process and grid-file based filtering phase in the trajectory join processing generate duplicated track identifier pairs, it eliminates the need of complex spatial indexing which is difficult to implement on GPUs in general. However, as mentioned at the end of Section 3 and discussed in Section 5.3, the duplications do impose some memory pressure on GPUs. We are also in the process of exploring multi-level grid-file structures to reduce or eliminate the duplications.

5. EXPERIMENT AND RESULTS

5.1 Data and Experiment Setup

To test the feasibility of the system design and the performance of the implemented modules, we have used the Microsoft Research Asia T-Drive [1] and GeoLife [38] datasets. They are provided as two sets of text files with different structures. Following the architectural design introduced in Section 2, we transform each dataset into four arrays so that they can be efficiently streamed among disks, CPU memories and GPU memories. We have processed both datasets but will use the T-Drive dataset for visual exploration purposes (Section 5.2) and use the GeoLife dataset to test the performance of the GPU-based trajectory similarity query processing (Section 5.3). We have discretized the study region, i.e., (116.1000, 39.7000, 116.7553, 40.35530), into a 65536*65536 grid with a resolution of 10^{-5} degree (the maximum precision provided in the original datasets). As such, a grid cell has a spatial extent of 10^{-5} degree by 10^{-5} degree. The with and height of a grid cell are approximately 0.85 meter and 1.11 meter along the longitude (X) and latitude (Y) direction, respectively. All GPS point locations in the two datasets are aligned to grid cells. All experiments are performed on a Dell Precision T5400 workstation equipped with dual quadcore CPUs running at 2.26 GHZ with 16 GB memory, a 500GB hard drive and an Nvidia Quadra 6000 GPU device with 448 cores and 6 GB memory.

The trajectories in the GeoLife dataset are chunked based on the annotated travel mode labels that come with the dataset. We are particularly interested in the trajectories that are labeled as “walk” for trajectory similarity queries from a data management perspective as their bounding boxes are relatively small and have good filtering power. Two preprocessing steps are performed to make the trajectory similarity query feasible and interesting from a practical perspective. First, GPS points that are outside of the study area are removed. This results in 1,178,524 points out of the 1,440,823 points that are in the trajectories labeled as “walk”. The number of trajectories after this step is 3,245. Second, we have removed trajectories whose MBR areas are larger than 0.0001 square degrees in the study area, in addition to trajectories that have only one point (MBR areas are 0). This steps results in 2,341 trajectories. To better understand the distributions of the trajectories, we have plotted their MBRs in Fig. 5. Clearly, the majority of the selected trajectories are located in the Northern part of Beijing, especially in the areas that are close to Microsoft Research Asia office. These trajectories will be used for trajectory similarity queries and the results will be reported in Section 5.3.

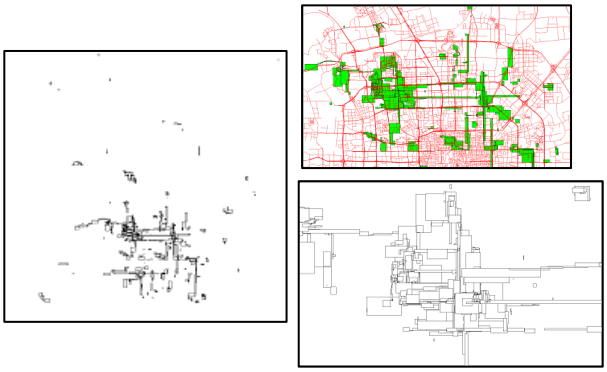

Fig. 5 MBR Plots of GeoLife Trajectories (left) and Two Detailed Views in Different Presentations (Right)

5.2 Visualization of Raw Trajectories and Gridded GPS Points

U^{2}STRA supports exporting internal representations of trajectory data into SQL statements which can be imported in the open source PostgreSQL database through the PostGIS plugin. Subsequently, these trajectory data can be exported to ESRI Shapefile format that is accepted by many GIS software (such as ESRI ArcGIS and QGIS) for visualization and visual exploration purposes. A snapshot of the T-Drive dataset by connecting two neighboring GPS location points as lines is shown in the left part of Fig. 5. At the city scale, the directly plotted GPS trajectories show the overall network topology of Beijing City reasonably well. A major problem of plotting all trajectories by linking two consecutive points is that, for trajectories with low sampling rates, the connected lines can deviate from true travel paths significantly. When plotted together, they are likely to clutter the whole display space without providing much useful information at finer scales, as shown in the right part of Fig. 5.

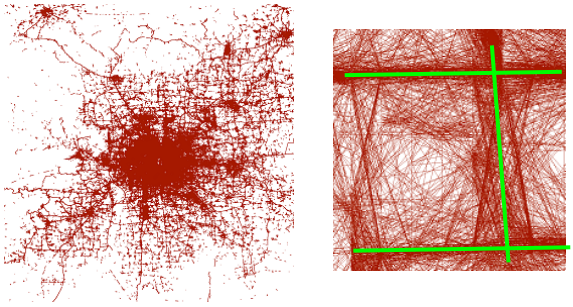

Fig. 6 Visualizing GPS Trajectories in GIS at the City Scale (Left) and Block Scale (Right)

An alternative solution is to plot the GPS points separately without connecting them. However, there are two major problems for the straightforward approach. First, when the numbers of points are large (e.g., millions to billions), drawing the points in a GIS environment is extremely slow due to large data volumes and graphics rendering overheads. Second, due to limited screen resolutions, points that are close to each other very often are overshadowed. Both of the problems prevent from seeing a clear picture of the structures that can derived from the underlying GPS data. As such, we reuse our grid-based spatial aggregation module to compute the numbers of GPS points that fall within a raster grid and output the aggregated results into an image. The result of aggregating 17,762,489 GPS points in the T-Drive dataset using an 8192*8192 grid is shown in Fig. 6. Compared with Fig. 5, plotting aggregated GPS points seems to be a better approach than plotting trajectories directly.

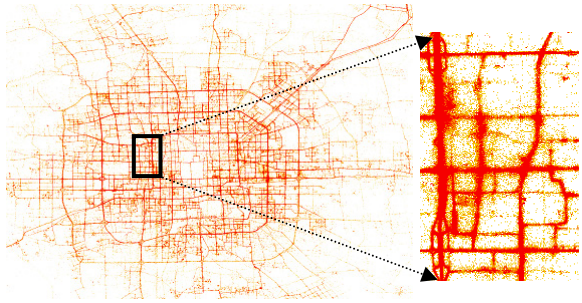

Fig. 7 Visualizing Aggregated GPS Point Locations

Our results show that U^{2}STRA is able to aggregate the nearly 18 million points in 47.25 milliseconds while it took 4,110.27 milliseconds for the same aggregation on a serial CPU implementation using STL (Standard Template Library [44]) with O2 optimization. Clearly an impressive 87X speedup has been achieved. While the performance of the serial CPU implementation is acceptable for this relatively small dataset, we expect the speedup will be much more desirable for large GPS datasets, such as OpenStreetMap Planet GPS point dataset with 2.77 billion GPS points (we are working on it actively to derive world-wide roadmap). Interested readers are further encouraged to access the link at [45] to our website to visualize the derived road map in a Web-based GIS environment. We have registered the aggregated raster with the OpenStreetMap data in the same area. An interesting observation is that, the derived road map from GPS locations is able to show some new roads that do not exist in the OpenStreetMap data yet. The results suggest that near real time taxi GPS trajectories can potentially be used to update city street maps at a much finer temporal resolution.

5.3 Results on trajectory similarity queries

In this set of experiments, we focus on the efficiency of the query processing on trajectory similarity join. Before discussing the performance, we provide an example from the GeoLife dataset in Fig. 8 to help understand the Hausdorff distance based similarity measurement better. In Fig. 8, the red dashed line is the longest among all the dashed lines which represent the shortest distances from all the points in the green trajectory to the blue trajectory. We have used this approach to verify the correctness of the implementation by looking into selected trajectory pairs.

We use a grid dimension of 8192*8192 for rasterization during the filtering phase although other grid dimensions are possible. To test the scalability of the GPU implementation, we have used different D values and report the results in Table. 1. For comparison purposes, we have also implemented the refinement phase in the query processing on CPUs using a single core (serial implementation). We did not implement the filtering phase on CPUs for two reasons. First, we are not aware of efficient open source implementations of main-memory based spatial indexing on CPUs and our GPU implementation, while has achieve impressive throughput, may not be efficient on CPUs due to the overhead of parallelization coordination. It would be inappropriate to simply serialize the

GPU implementation on CPUs and report its performance. Although we could have used external memory spatial indexing packages (such as libspatialindex [43]), it would not be fair to compare the performance directly as the gap between main-memory structures and external memory structures are well-known. Second, perhaps more importantly, the runtimes of the filtering phase is dominated by the refinement phase on both GPUs and CPUs. The runtimes of the filtering phase on CPUs do not affect GPU speedups significantly due to Amdahl's law [46]. The runtimes for of the refinement phase of the CPU implementation for different D values are also shown at the lower part of Table. 1.

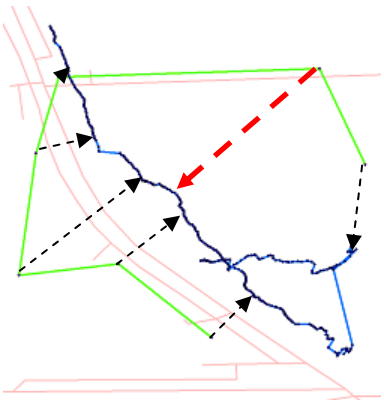

Fig. 8 A Real Example Showing Hausdorff Distance between Two trajectories Identified During the Experiments

From Table 1 we can clearly see that our GPU based implementation has achieved 35-40X speedup on distance computation in the refinement phase. Even the runtimes of the filtering phase on CPUs are excluded, the GPU implementation still achieves overall 25-35X speedup (last row of Table 1). However, the experiments also confirmed the analysis on the GPU memory pressure imposed by the simple grid-file based filtering framework. We can also see that, as D increases, the runtimes of the filtering phase weigh higher which is the primary reason that the speedups (SP-RT and SP-Overall) get lower. Although our GPU implementation is capable of handling up to NCP=257 million paired cell identifiers for $D=500*10^{-5}$ degree (as shown in Table 1), it runs out of GPU memory for $D=750*10^{-5}$ degree where NCP=379 million which clearly indicates that the GPU implementation is memory bound. From Table 1 we can see that both NCE (number of rasterized cells of expanded MBRs) and NCP (number of paired cell identifiers) grow almost linearly with D but NCE has a higher slope. This can be explained by the fact that the majority of trajectories in walk mode follow street segments which are either horizontal or vertical in most cases in Beijing. As such, the areas of expanded MBRs of these trajectories, which are proportional to NCE, grow mostly linearly except those with significant portions of perpendicular turns (which do happen). The reason that NCP has a lower grow slope can potentially be attributed to the fact that only the expanded part of MBRs are paired with additional cells while the base part does not. In another word, NCP grows with Σ(area(Expanded MBRi)-area(MBR)) which is less than Σ(area(Expanded MBRi)).

Table 1 Numbers of Rasterized and Paired Cell Identifiers, Runtimes and GPU Speedups from the Experiments

D ($*10^{-5}$ °)	50	100	200	500	750
NCO ($*10^6$)					2.441
NCE ($*10^6$)	4.324	6.934	14.35	54.16	107
NCP ($*10^6$)	38.34	56.06	100.2	257.6	379
NUP ($*10^6$)	0.095	0.120	0.171	0.318	-
GPU-FT (ms)	277.8	418.9	779.6	2150	-
GPU-RT (ms)	2124	2505	3235	5220	-
GPU-TT (ms)	2402	2924	4015	7370	
CPU-RT (s)	83.50	96.30	120.3	181.9	
SP-RT(X)	39.3	38.4	37.2	34.8	
SP-Overall (X)	34.8	32.9	30.0	24.7	

Notes: NCO: # of rasterized cells of original MBRs; NCE: #of rasterized cells of expanded MBRs; NCP: # of paired cell ids: NUP: #of unique trajectory pairs whose MBRs are within D; GPU-FT: GPU filtering time; GPU-RT: GPU refinement time; GPU-TT=GPU-IT + GPU-FT + GPU-RT; CPU-RT: CPU refinement time; SP-RT(refinement speedup) = CPU-RT*1000/GPU-RT; SP-Overall (Overall Speedup) =CPU-RT*1000/GPU-TR.

Another observation from Table 1 that is worthy of discussion is the ratio between NCP and NUP (number of unique trajectory pairs whose MBRs are within D) is in the order of 400 to 800 which grows sub-linearly as D grows. The ratio might be too high with respect to efficient filtering. We have decided to use a large grid file (8192*8192) for the filtering phase which is effective in reducing computing Hausdorff distances. The reduction rate can be computed as NUP*2/(N*N) where N is the number of trajectory tracks. When plug in N=2,341, the reduction rate varies from 3.47% to 11.6%. We are considering use smaller grid files to reduce NCP and hence memory requirement at the cost of increasing the NUP and hence the workload in the refinement phase. It would be interesting to derive a quantitative analytical framework to seek optimal parameters with respect to the grid file sizes. Another possible direction is to use multi-level grid files to reduce NCP directly as cells that are matched at the upper levels do not need to be matched in the lower levels. This is left for future work.

6. CONCLUSION AND FUTURE WORK

In this study, we have explored the research opportunity in using massively data parallel GPGPU technologies for trajectory data management which is becoming important due to the popularities of GPS and other locating and navigation devices. In particular, we have developed the U^{2}STRA prototype system to perform parallel aggregations to understand the overall patterns of GPS point locations in trajectory datasets and process similarity trajectory queries based on the Hausdorff distance. We have also developed a practical in-memory data layout schema that has low memory footprint and is cache friendly, in addition to supporting flexible data organization and retrieval. The experiments have shown that spatial aggregations of nearly 18 million GPS point locations in the T-Drive dataset has achieved 87X speedup compared with a serial CPU implementation using STL and 25-40X speedup on trajectory similarity queries over an optimized serial CPU implementation in the refinement phase that requires intensive distance computation. The simple grid-file based spatial indexing also provides a solid foundation for future improvements.

For future work, first of all, we plan to develop a more efficient parallel data structure on GPUs for filtering in trajectory query processing by exploring multi-level grid-files

and other options. Second, we want to investigate the suitability of GPGPU computing technologies for trajectory data cleaning and segmentation which are usually also computing intensive. Third, our U^{2}STRA system currently only supports Hausdorff distance based similarity query and we plan to investigate on more measurements of similarity and support different types of trajectory queries. Finally, we plan to test the system on larger dataset (such as OpenStreetMap Planet GPS point dataset) and develop techniques to reduce GPU memory capacity bottleneck.

REFERENCES

1. User Guide of T-Drive Data. http://research.microsoft.com/pubs/152883/User_guide_T-drive.pdf
2. http://wiki.openstreetmap.org/wiki/Planet.gpx
3. Reades, J., Calabrese, F., et al. 2007. Cellular Census: Explorations in Urban Data Collection. Pervasive Computing, IEEE 6(3): 30-38.
4. Calabrese, F., Colonna, M. et al. 2010. Real-Time Urban Monitoring Using Cell Phones: A Case Study in Rome. IEEE Transactions on Intelligent Transportation Systems 12(1): 141-151.
5. Mokbel, M..F, Ghanem, T.M., Aref, G., 2003. Spatio-Temporal Access Methods. IEEE Data Eng. Bull. 26(2): 40-49.
6. Nguyen-Dinh, L., Aref, G., Mokbel, M..F, 2010. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). IEEE Data Eng. Bull. 33(2): 46-55
7. Güting R.F., and Schneider, M., 2005. Moving objects databases. Morgan Kaufmann. ISBN-13: 978-0120887996
8. Deng, K., Xie, K., Zheng, K. and Zhou, X., 2011. Trajectory Indexing and Retrieval. In Yu Zheng and Xiaofang Zhou (eds) Computing with Spatial Trajectories. Springer.
9. Cudre-Mauroux, P., Wu, E. and Madden, S. (2010). TrajStore: An adaptive storage system for very large trajectory data sets. Proceedings of IEEE ICDE Conference.
10. He, B. S., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q. and Sander, P. V. (2009). Relational Query Coprocessing on Graphics Processors. ACM Transactions on Database Systems 34(4).
11. Bakkum, P. and Skadron, K. (2010). Accelerating SQL database operations on a GPU with CUDA. Proceedings of GPGPU workshop, 94-103.
12. Zhang, J., Gong, H. et al 2012. U^{2}SOD-DB: A Database System to Manage Large-Scale Ubiquitous Urban Sensing Origin-Destination Data. Proceedings of ACM SIGKDD Workshop on Urban Computing.
13. 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
14. Zhang, J., You, S. and Gruenwald, L. (2012). High-Performance Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs. Technical Report. Online at http://www-cs.ccny.cuny.edu/~jzhang/papers/aggr_tr.pdf.
15. Zhang, J., You, S. and Gruenwald, L. (2012). Speeding Up High-Performance Spatial Join Processing on GPGPUs with Applications to Large-Scale Taxi Trip Data. Technical Report. Online at http://www-cs.ccny.cuny.edu/~jzhang/papers/nnsp_tr.pdf.
16. Sergio Ilarri, Eduardo Mena, Arantza Illarramendi. 2010. Location-dependent query processing: Where we are and where we are heading. ACM Computing Surveys (CSUR). 42(3) 1-73
17. Giannotti, F., Nanni, M., et al., 2011. Unveiling the complexity of human mobility by querying and mining massive trajectory data. The VLDB Journal 20(5): 695-719.
18. Renso, C., Baglioni, M., et al, In Press. How you move reveals who you are: understanding human behavior by analyzing trajectory data. Knowledge and Information Systems: 1-32.
19. Güting, R., Braese, A., et al, 2009. Nearest Neighbor Search on Moving Object Trajectories in SECONDO. Proceedings of SSTD.
20. Düntgen, C., Behr, T. and Güting, R. (2009). BerlinMOD: a benchmark for moving object databases. The VLDB Journal 18(6): 1335-1368.
21. http://m-atlas.eu/
22. http://www.postgresql.org/
23. http://postgis.refractions.net/
24. Trajcevski, G., Ding, et al., 2007. Dynamics-aware similarity of moving objects trajectories. Proceedings of ACM-GIS.
25. Somayeh, D., Robert, W. and Patrick, L., 2009. Exploring movement-similarity analysis of moving objects, ACM. SIGSPATIAL Special 1,11-16.
26. Chen, Y. and Patel, J. M., 2009. Design and evaluation of trajectory join algorithms. Proceedings ACM-GIS.
27. Gunopulos, D. and Trajcevski, G., 2012. Similarity in (spatial, temporal and) spatio-temporal datasets. Proceedings of EDBT.
28. Adelfio, M., Nutanong, S. and Samet, H., 2011. Similarity search on a large collection of point sets. Proceedings of ACM-GIS.
29. Hennessy, J.L. and Patterson, D. A, 2011. Computer Architecture: A Quantitative Approach (5th ed.). Morgan Kaufmann
30. http://en.wikipedia.org/wiki/Online_analytical_processing
31. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690
32. Garcia, V., Debreuve, E. and Barlaud, M., 2008. Fast k nearest neighbor search using GPU. Computer Vision and Pattern Recognition Workshops, 2008.
33. Cayton, L., 2010. A Nearest Neighbor Data Structure for Graphics Hardware. Proceedings of ADMS.
34. Pan, J. and Manocha, D., 2011. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. Proceedings of ACM-GIS.
35. Pan J. and Manocha, D., 2012. Bi-level Locality Sensitive Hashing for k-Nearest Neighbor Computation. Proceedings of IEEE ICDE Conference, 378-389
36. Kruls, M., Skopal, T., Lokoc, J. and Beecks, C. 2012. Combining CPU and GPU architectures for fast similarity search. Distributed and Parallel Databases 30(3): 179-207.
37. Kato, K. and Hosino, T. 2012. Multi-GPU algorithm for k-nearest neighbor problem. Concurrency and Computation: Practice and Experience 24(1): 45-53.
38. Michael, G. and David, B. K., 2010. Understanding throughput-oriented architectures. CACM 53(11): 58-66.
39. http://research.microsoft.com/pubs/152176/User%20Guide-1.2.pdf
40. http://www.opengeospatial.org/standards/sfs
41. Jacox, E. H. and Samet, H. (2007). Spatial join techniques. ACM Transaction on Database System 32(1).
42. http://thrust.github.com/
43. http://libspatialindex.github.com/
44. http://www.sgi.com/tech/stl/
45. http://geoteci.engr.ccny.cuny.edu/geoteci/tdrive_bj.html
46. http://en.wikipedia.org/wiki/Amdahl's_law