GTS: A GPU-accelerated Trajectory Storage System Supporting Multiple Spatial Queries

Zhang Bowen
Department of Computer Science and Engeneering
Shanghai Jiao Tong University
Shanghai, China 200240
Email: zbw0046@sjtu.edu.cn

***Abstract*—As the development of smart devices equipped with GPS, here comes a large amount of trajectory data, implying many useful information about our daily life. This call for a trajectory storage system able to handle mass of various kinds of queries efficiently. GPU, which has been widely equipped on computers, can accelerate queries by handling them in parallel. However, existing GPU-accelerated trajectory storage systems are optimized for specific kind of query, unable to support wide range of applications. To solve this problem, we propose a storage system optimized for the features of GPU, supporting multiple kinds of queries. We exploit the potential opportunities of PR-quadtree in adapting to different strategies for pruning, designing an adaptive storage component. To make full use of the parallel power of GPU, we develop a query engine optimized with load-balancing, coalesce memory accessing and less data transferring. We evaluate our system on trajectory dataset of cars in Shanghai, which shows our system able to conduct three basic kinds of queries efficiently. Moreover, our system speed up about 10x for EDR measured top-k similarity query than implementing on CPU, demonstrating that GPU can be used to accelerate queries on large-scale trajectory data.**

I. Introduction

1.background

As the number of mobile location devices (like smartphones, cars, public bicycles, etc.) grows, increasing amount of spatial-temporal sequential data, i.e. trajectory data are collected from the every corner of the world. For example, 80GB trajectory data are generated during July 1, 2013 and April 30, 2014 just from xxx cars in Shanghai, which contains more than xxx sample points. There are many interesting and valuable information in these data to be exploited, such as route planing[...], traffic prediction[...], travel time prediction [1] etc.. This calls for a solution of efficient trajectory storage and retrieval. Recent years various kinds of spatial indexes and in-memory storage are proposed to increase throughput and decrease latency. Meanwhile, having witnessed the growing application of GPU in large-scale data processing, it is a popular choice that use GPU to answer mass of queries in parallel [2][...].

(picture of two kinds of query)

2.problem description

Thousands of queries are produced everyday in these applications. In this situation, query delay is significantly important for user's experience. However, existing trajectory storage systems are almost optimized for specific kind of query, causing that applications invoking different kinds of query lose efficiency when running on them. Although a mass of query algorithm are designed for different purposes recently, they can be concluded in two main types: range-based query and trajectory-based query, which can be represented by range query and top-k trajectory similarity query respectively. [3] This two kinds of query are both indispensable for many kinds of trajectory applications. For example, xxx need xxx. It is not wise to trivially add an additional query module to support the queries not included in their original optimization consideration because of the low pruning performance caused by mismatching index not optimized for this kind of query.

3.challenge of solving problem

To solve this problem, it is intuitive to set two database for different kind of query respectively. Unfortunately, this will cause unneccessary space comsumption because of duplicate data are stored, which is harmful for memory efficiency. Considering that less space consumption means a mass of benefits for trajectory data mining, such as more trajectories can be persisted in main memory as training set to improve accuracy, it's neccessary to find a space-saving solution.

4.existing work about this problem

i.SharkDB propose a column-based data structure to store trajectory data leverageing the time-aggregation of queries. But it can't support trajectory-based query, such as trajectory similarity query because column-based storage makes it hard to access the whole trajectory sequence fastly, which is an important step in similarity query. Besides, the parallel query algorithm that SharkDB develope is only for thread pool, which can't easily migrate to GPU.

ii.TKSimGPU propose a query algorithm that implemented on GPU which is designed for trajectory similarity query. However, the similarity measurement which TKSimGPU use is outdated, because it doesn't take local time shifting into concern, which has been proved a significant issue for similarity measurement of trajectory(cited by DTW,EDR,...). Moreover, range query is not supported efficiently by TKSimGPU because of non-adaptive grid.

5.my idea

In this work, to solve this problem, we design and implement the GTS (GPU-accelerated Trajectory Storage) system, which can handle both range query and local time shifting based top-k similarity query on historical trajectory data.

Recent years many similarity measurements based on local time shifting are proposed and be verificated outperforming traditional Euclidean distance based measurements[][]. So it is important for trajectory storage system to support this kind of query. We design a single storage component supporting pruning on both two kinds of query with no duplicated data, to save memory space. Moreover, our design is friendly with GPU, which means we can make use of parallel power of it to accelerate query engine. It is worth mentioning that we are the first one to design top-k similarity query algorithm for GPU supporting local time shifting.

6.challenge of idea

However, to implement this idea, there are some main challenges in both storage designing and query algorithm designing. (i) Firstly, it is difficult to design index optimized for both two types of query at the same time. In existing work, people use location-based index which is metric to handle range query, for example, kd-tree. However, local time shifting based trajectory similarity query, such as [4], calls for pruning methods which is based on inequation derived from the properties of similarity measurements, which is usually not metric.

(ii) Secondly, many issues should be concerned to improve performance when accelerating the queries with the help of GPU. For example, local time shifting based similarity measurements, such as EDR, are not trivial to be transplanted into GPU. For traditional similarity query, parallelization of query algorithm is easy by dividing whole trajectory into segments first and then assigning each segment to a thread to handle. However, different from this, calculation of EDR is correlated with all points of trajectory, which means a global optimal alignment, making the fine-grain parallelization more difficult. Apart from this, the memory accessing pattern in the process of EDR is discrete when maintaining state matrix. This wastes the bandwidth of GPU's global memory and thus hurts the query performance.

(I'm finding more challenge...)

7.main issues

i. To address the challenge (i), we propose a novel trajectory storage component based on quadtree. We find that data structures used in pruning process of EDR is similar with grid, which has been widely used as index for GPU-accelerated range query. On the basic of this observation, we subtly adopt the characters of Morton encoding and PR-quadtree to design an index supporting pruning both on range query and top-k similarity query.

ii. To address the challenge (ii), we design the query algorithm which is load balancing and has the coalesce accessing pattern to avoid the performance loss caused by improper memory accessing pattern. For top-k similarity search, we leverage the potential independence of the procedure to design a scalable task division strategy considering both coarse-grain and fine-grain parallelism to make full use of GPU's parallel power. Otherwise, we design a special placement organization of elements in state matrix in GPU global memory, satisfying the requirement of coalesce accessing.

8.contributions

- We propose a GPU-accelerated in-memory trajectory storage system which support both range query and local time shifting based similarity query.
- We design a GPU-friendly storage component satisfying the requirement of pruning on both two kinds of query.
- We design a novel solution of accelerating local time shifting based similarity query on our storage component by exploiting GPU's parallel power.
- We deploy our system on server, evaluate the performance, and prove that our system get the 3x higher performance than state-of-the-art system.

9.organization

The rest of paper is organized as follows. In Section 2 we review the related work. After that, in section 3 we briefly introduce the architecture of our system. We propose our design of storage component in section 4. Query engine is described in detail in section 5 with three kinds of query algorithm. In section 6, we do an evaluation of our system and results are reported. Section 7 concludes the paper.

II. Background

We first give the preliminaries. Then we introduce some background knowledge of GPU used in our design.

A. Problem Definition

In this section, we propose some basic definition of trajectory and formulation about our problem. We first define some elements of trajectory.

Definition 1 (sample point): A sample point $p = (x, y, time)$ is a three-dimensional data which include spatial information represented by (x, y) and time stamp $time$. For simplicity, we assume that all sample points' coordinates have been transfered to Euclidean plane.

Definition 2 (trajectory): A trajectory of the object $t = \{p_1, p_2, \ldots, p_n\}$ is a sequence of sample points, where n is the length of this trajectory. To make the trajectories meaningful, we raise a regulation that the delta of timestamp between two consecutive sample points should be always within 30 minutes.

Given a large set of trajectories $T = t_1, t_2, \ldots, t_{|T|}$, our goal is answering all kinds of queries over them. As we mentioned in Section 1, they can be classified into range query, k-nearest neighbor point query and top-k trajectory similarity query. Here we formulate this three kinds of query.

Definition 3 (query): A query $q = (kind, cond, T)$ aims to find a set of trajectories or segments as results S from trajectory dataset T which satisfy query conditions $cond$, where $kind$ is the notation of the kind of the query.

For range query, the condition usually includes a bounding box $B = [xmin, xmax, ymin, ymax]$ and a time range $[time_s, time_e]$. A trajectory t is said to satisfy range query if there exist at least one sample point $e.p_1$ is included in the bounding box and its time stamp is within time range, which can be represented as $\exists p \in t, (xmin \leq p.x \leq xmax) \wedge (ymin \leq p.y \leq ymax) \wedge (time_s \leq p.time \leq time_e)$.

Top-k trajectory similarity query retrieves a set of k trajectories $R_{sim} = \{t_1, t_2, \ldots, t_k\}$ which are most similar with given trajectory t_q. There are many metric about trajectory similarity, and it's widely accepted that EDR distance [4] which can handle local time shifting may be a good choice. So we use EDR distance as our similairity metric.

Definition 4 (EDR distance [4]): Given two trajectories $t_1 = \{p_1, p_2, \ldots, p_{n1+1}\}, t_2 = \{p_1, p_2, \ldots, p_{n2+1}\}$, the EDR distance between them is calculated by:

$$EDR(t_1, t_2) = \begin{cases} n & \text{if } m = 0 \\ m & \text{if } n = 0 \\ \min\{EDR(Rest(t_1), & \\ Rest(t_2) + subcost), & \\ EDR(Rest(t_1), t_2) + 1, & \\ EDR(t_1, Rest(t_2)) + 1\} & \text{otherwise} \end{cases}$$

where $subcost = 0$ if $dist(t_1.p_1, t_2.p_1) \leqslant \varepsilon$ and $subcost = 1$ otherwise, and $Rest(t) = \{t.p_2, \ldots, t.p_{n+1}\}$, noting that ε is a threshold set by users.

B. GPU computing using CUDA[]

We design GTS based on CUDA proposed by Nvidia, a programming framework for GPU computing. In this part we introduce some background of GPU and basic concept in CUDA. GPU follows Single Intruction Multiple Data (SIMD) parallel model, indicating all of the cores in an Stream Microprocessor (SM) executing the same instruction on different data at the same time. There are tens of SMs in one GPU, each of which has tens of cores, forming a two-level parallel architecture. In CUDA, this architecture is reflected in the division of grid, block and thread. When one program, called kernel, is loaded in CUDA, a grid is generated, including mass of blocks. The block contains a fixed number of threads, which runs on an SM of GPU. Threads in the same block can share a high speed but small volume local memory on SM, but threads in different blocks can only use slow GPU global memory to exchange information, requiring us to divide the task properly into different blocks.

There are three main issues when using GPU. First, accessing pattern is a main issue when accessing global memory. When hundreds of threads in one block accessing data from global memory, if the data each thread needs are nearby, GPU will coalesce hundreds of accessing request to several continuous accessing, improving the bandwidth for tens of times. For example, if thread 1,2,..,32 access the data at offset 33,34,...,64 respectively, 32 data loading transitions will be combined to one single transition. Obviously it requires the data accessing by threads in a block are stored continuously. Second, the load balancing makes sure that all cores of GPU are not idle, leading to a high throughtput. It means we should assign almost equal amount of computation on one thread or one block. Third, PCI-E interface, the bridge between main memory and GPU, has much lower bandwidth than global memory and local memory. So the data transferring between main memory and GPUshould be reduced as much as possible to avoid the performance loss. These three issues raise a challenge for GTS to get a high speedup.

III. System Architecture

A. Overview

Our system includes two parts, storage component and query engine. As the Figure 1 shows, raw trajectories are split into sub-trajectories according to quadtree index, with the corresponding sequence of cell stored as cell-based trajectories. After raw trajectories are loaded into storage component, thousands of queries are handled in query engine, with the acceleration of GPGPU, outputing query results. It's worth noting that our system is designed for query-only applications. We then introduce storage component and query engine briefly.

B. Storage Component

In the design of storage component, we exploit the potential oppotunities of quadtree with mentor encoding [5] with the consideration of GPGPU. We divide all trajectories into sub-trajectories with small cell, after which each cell in quadtree is related to a set of sub-trajectories. To rebuild the whole trajectory from sub-trajectories easily, we propose a data structure named "cell-based trajectories" for each trajectory, storing the cell's identification it goes through, according to its original order. Sub-trajectories and quadtree are stored in in host memory, which is usually large enough to store the whole data. And cell-based trajectories are stored in GPU global memory, for faster parallel pruning on similarity query. The detail of storage component will be introduced in Section IV.

C. Query Engine

Our query engine is designed for handling thousands of queries including three basic kinds in parallel, optimizing for executing time per query. For different kinds of query, we first adopt different pruning strategies to reduce unneccessary computation by help of index. After that, we divide the all the queries into tasks at both coarse-grained level and fine-grained level, according to the CUDA's programming framework. We then transmit as less data to GPU, which executes produced tasks in parallel and output the query results. We will illustrate query engine in Section V.

IV. GPU-friendly Storage Component

In this section, we first introduce the index based on Mentor encoded quadtree. After that, we propose the benefits of Mentor encoded quadtree in supportting three kinds of queries, considering the characters of GPGPU computing mentioned before. At last, we propose the construction of our index.

A. Storage Design

Index performs as an important role in storage system by pruning unneccessary accessing of data. However, existing works about index on trajectories can not be easily used to solve our problem. First, in previous researches about trajectory storage system, different kind of query need different type

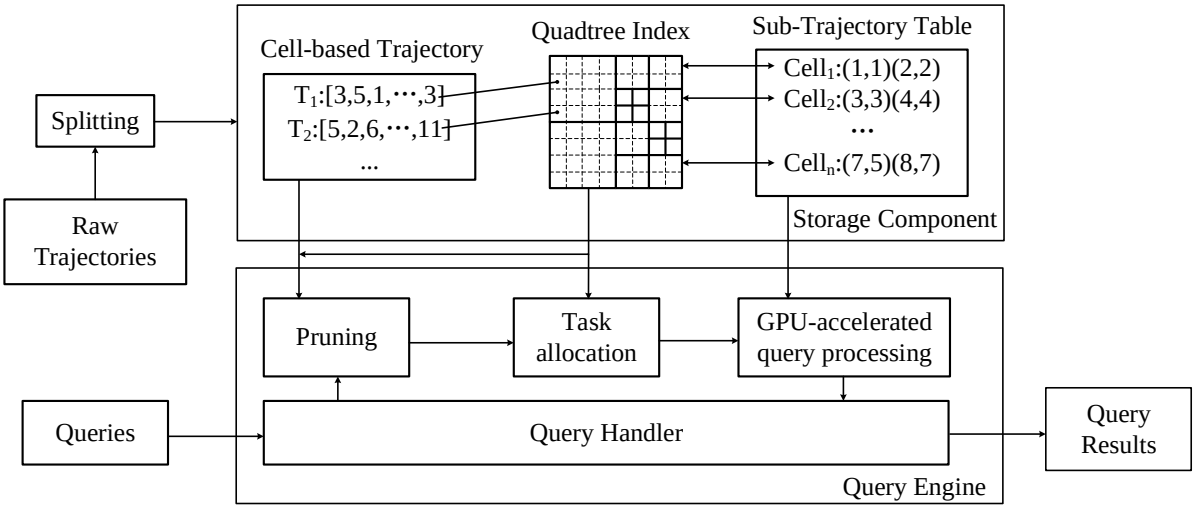


Fig. 1. System Architecture

of index, because of different characters of pruning strategies. Pruning on range query and k-nearest neighbor point query concern more about the relation between one query location and a trajectory. On the other hand, pruning on top-k similarity query concerns about the relation between two trajectories, which are sequences of locations, make it differentiated from two kinds of queries mentioned. Second, as mentioned in background, programming framework of CUDA raises a mass of claims about storage.

Fixed grid index can be used to solve the first challenge. An example of fixed grid is shown in middle in storage component of figure 1 . Each cell has an identifier and stores the pointers linking to the sub-trajectories it contains. For top-k similarity query, an lower bound of similarity distance can be derived by a histogram method [4], which plays an important role in pruning strategy. In this histogram method, given the whole plane $[(x_{min}, x_{max}, y_{min}, y_{max})]$, it is divided into τ_x intervals in x axis and τ_y intervals in y axis, forming bins $\{(x_i, x_{i+1}, y_j, y_{j+1}) | 1 \leq i \leq \tau_x, 1 \leq j \leq \tau_y\}$. For each bin, the number of points of each trajectory falling in the area of this bin is recorded. The trajectory histogram is then formed with all the bins. After that, the frequency distance between trajectories is defined on their histograms, which can generate the lower bound of EDR. Fixed grid index can store all the information in histogram. For example, in figure 2, with the trajectory t represented as line and sample points represented as points, a histogram is generated by counting the number of points in each cell, as the middle shows. The histogram information can be represented in grid like the right picture. Considering that fixed grid has been widely used as index in spatial database for range query, it is an valuable oppotunity that use fixed grid to deal with both two kinds of queries.

However, because of the heterogeneous distribution of trajectories, fixed grid suffers from problems of load balancing when used as index for GPU-accelerated query. The number of sample points in each cell will vary largely. For example, one

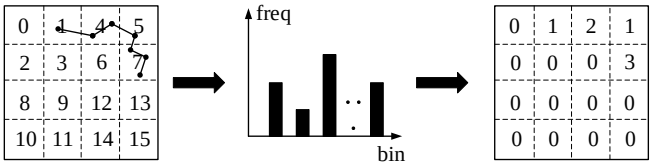
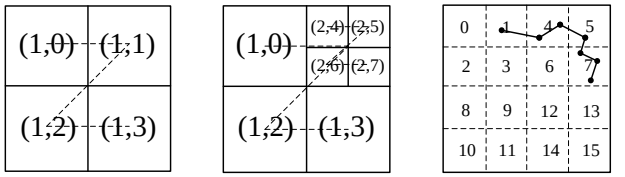


Fig. 2. Example of the trajectory in area (left), the histogram of this trajectory (middle) and the grid that can represent the histogram's value (right)

cell may contains too many sample points. In this situation, if we assign this cell to a thread, the other threads handling less points have to wait for this thread, which may reduce the speedup ratio of our system.

We adopt the idea of PR-quadtree [5] to overcome this challenge. PR-quadtree is an index used to handle streaming k nearest neighbor (kNN) queries for objects, achieving the load balancing on GPU. However, it can not serve for pruning of similarity query because nodes of PR-quadtree is not with the same geographical size. To apply the advantages of PR-quadtree to our system, we develope GT-quadtree based on fixed grid index by absorbing some useful features of PR-quadtree. It reaches the goal of achieving load balancing on GPU, and meanwhile remains the function of serving as the index of similarity query. In GT-quadtree, each node is corresponding to different number of cells of fixed grid, overlaping different area in geographical plane. As figure (x) shows, the root is corresponding to the whole plane. Except for root, one and another levels of nodes are generated recursively by dividing areas of their parent into four quadrant to make sure the number of points within each node's area is all less than a user specified parameter vol_{max}. Also, if the number of points in a node is less than vol_{max}, it would not be divided anymore and regarded as leaf node. GT-quadtree is built based on fixed grid, with each leaf node containing all the cells of grid within its spatial area. Morton encoded identifiers are assigned to cells of fixed grid and nodes at different levels

of GT-quadtree, as figure 3 shows, noting that the node i in j level is represented as (j, i). With the help of Morton encoding, we can traverse from a node to its parent within a bitwise operation in GT-quadtree and vice versa. For example, in case showed in figure 3, the parent of node $(2, 4)$ in figure 3(b) is $(1, 1)$ in figure 3(a). This process is done by profroming a two-bits right shift on $100(4)$, getting $1(1)$.



(a) Morton encoding in level 1 (b) Morton encoding in level 1 and 2 (c) Morton encoding in grid of GT-quadtree

Fig. 3. An example of Morton encoded GT-quadtree

We designed sub-trajectories arrays storing the original geogrophical information of trajectories, making GT-quadtree meet with the memory access pattern required by GPU. Sub-trajectories array contains the sample points whose locations are within a given cell in GT-quadtree's fixed grid. These points are organized as the order of time. Meanwhile, points belong to the same trajectory are distributed together. For example, if there are sample points $\{p_{1,1}, p_{1,2}, ..., p_{1,m}\}$ and $\{p_{2,5}, p_{2,6}, ..., p_{2,n}\}$ from trajectory t_1 and t_2 in cell 15, the content of sub-trajectories array of cell 15 will be $\{p_{1,1}, p_{1,2}, ..., p_{1,m}, p_{2,5}, p_{2,6}, ..., p_{2,n}\}$. This arrangement benifits the query performance because when the points of one cell are transferred into GPU they are located together in global memory, which satisfies the condition of coalesce accessing when GPU cores fetching data from global memory. With Morton encoding of identifiers of nodes, cells within the same node are stored continuously, indicating points within the same node are also stored continuously. **This benefit will be illustrated in detail when we introduce query engine. (Add a figure to illustrate this)**

Operation of accessing to the whole trajectory is neccessary for similarity query based on local time shifting. To satisfy this requirement, we create the cell-based trajectory list to obtain the whole trajectory with the help of sub-trajectories arrays and fixed grid of GT-quadtree. The list contains the cell-based trajectories, each of which is transformed from the original trajectory by recording all cells' identifiers it goes through as a sequence. For example, in figure 3, the trajectory can be transformed into its cell-based trajectory $[1, 4, 4, 5, 7, 7, 7]$. After this transformation, we can fetch the whole trajectory by reading the sample points from sub-trajectories array of cells in cell-based trajectory one after another. We can also generate the frequency vector of a trajectory quickly from corresponding cell-based trajectory, which plays an important role in pruning for similarity query.

B. Construction

(This part differ very small)

Storage component construction starts initially with an empty fixed grid covering the whole spatial plane. We first set the length of each cell is set to m times of specified noise threshold of EDR distance ϵ. It make the cell equivelent as two-dimensional bin. Then we build a grid containing 4^m cells, to guarantee that we can build a quadtree on it. Identifiers of cells are assigned according to the rule of Morton encoding, as figure 3(c) shows.

After that, we split the input trajectories into sub-trajectories. For each trajectory, we allocate sample points to corresponding cell of the grid, in order of cell's identify one by one. Sample points within the same cell are stored continuously in sub-trajectory array, with the offsets of each trajectories are marked in the cell. In this process, we record the sequence of cells allocated, generating the cell-based trajectory. We transfer all cell-based trajectories to GPU global memory, forming the cell-based trajectory list.

Based on this fixed grid, similar with the method in [5], we build a new quadtree as our final index, in which the number of each cell not exceeding a given threshold vol_{max}, assuring load balancing when dividing range query into multiple tasks. In the process of building quadtree, we start with creating an empty root node which covers all of the cells as the root of quadtree, identified as (0,0). Then we calculate the volumes for all nodes in quadtree, which indicates the number of sample points the node contains. We say a node contains a sample point if this point is within the cell which this node covers. Then, nodes whose volume is larger than vol_{max} will be divided into four quadrants, forming the children nodes in next level, meanwhile other nodes are labeled as leaf nodes, containing less than vol_{max} points. This procedure will be invoked recursively until there is no node to be divided. For example, in figure 3, node (1,1) in 3(a)is divided into (2,4), (2,5), (2,6) and (2,7) in 3(b).

V. GPU-accelerated Query Engine

In this section, we introduce our query engine working on CPU-GPU hybrid environment. The goal of query engine is to handle both range queries and top-k similarity queries from thousands of clients and get speedup through executing tasks on GPU. For this two kinds of queries, we design parallel query algorithms respectively, including task division strategy and query procedure.

A. Range Query

(This part is simple and similar with work in other works, so use several sentence to describe.)

For range queries, we consider the combination of two levels of parallel, containing parallel within query and parallel among queries to fit with the two-level parallelism model of CUDA programming. We achieve this in two step.

First, we filter sample points not within query ranges quickly through GT-quadtree. In this step, we first extract query's ID and their query ranges. Then, nodes in GT-quadtree whose cells are overlapped by the ranges of queries are retrieved and noted as candidate nodes, forming a mass of

$\langle node, query \rangle$ pairs. It can be easily seen each candidate node should indeed be refined, for there exists at least one cell in it may include sample points within the result of range query. To prepare for the refine phase executed by GPU cores, we transfer trajectory data within retrieved node from host memory to GPU immediately after the node is marked as candidate, leaving CPU continue traversing the GT-quadtree simultaneously. To get around duplicated data transfering, we maintain a table recording the node which has been in GPU global memory and corresponding pointer so as to check whether node has been transfered to GPU.

At the second step, we refine the candidates on GPU in parallel and get the final results. Each thread block is assigned with a $\langle node, query \rangle$ pair, finding out sample points in node within query's range. In each thread block, to guarantee coalesce accessing, sample points are verified in loops. As mentioned in GT-quadtree, sample points of the same node are stored continuously in global memory. For this reason, we propose an strategy that one thread looks for one point in each loop and output whether it is within range according to thread ID until all points within the node are verified. In this strategy memory request of the threads are nearby, leading to coalesce accessing. For example, in the first loop, thread 0 handles point 0, thread 1 handles point 1, ..., and so on. In addition, reminding that each node in GT-quadtree contains sample points no more than vol_{max}, the load balance is achieved in this step.

B. Top-k Similarity Query

The goal of our solution is generating the result of top-k similarity queries in a short time, leveraging the parallel power of GPU. We use EDR as the similarity measurement, which is popular and performs well in most of recent works. [][] Our work can be migrated to other measurements whose calculation is based on dynamic programming algorithm, such as LCSS[]. Considering that GPU efficiency is low if the task load is small, we adapt the algorithm proposed by [4] to fit with GPU architecture and design a filter and refine scheme, which is shown in Algorithm 1. The initial idea of this scheme is filtering some trajectories which can be assured not appearing in the result to avoid unneccessary expensive EDR computing.

First, for each query, we use cell-based trajectories to calculate the lower bounds of EDR between query trajectory and trajectories in storage and add all trajectories to candidate set (line 3-5). After that, we find η trajectories from candidate set whose lower bounds are the lowest, calculating the real EDR between query trajectory. Obtaining the upper bound of current top-k EDR distance of trajectories efficiently is a frequent operation for pruning, so a size-k priority queue of real EDR distance is maintained to handle this operation. (line 7-8). We then prune trajectories in candidate set whose lower bound is bigger than upper bound of current top-k EDR distance safely because we can assure they will never be chosen in following iterations (line 9). We repeat this process of choosing top-η trajectories from candidate set and pruning until candidate set is empty. Finally trajectories in the priority queue are results of this top-k similarity query.

Algorithm 1 Top-k Similarity Query

Input:
Query Trajectory Set $Q = \{q_1, q_2, ..., q_M\}$; Historical Trajectory Data $D = \{d_1, d_2, ..., d_N\}$; Parameter k,η;

Output:
Result list for each query $Result$

1: $Result \leftarrow list(array[k])$
2: **for** $q \in Q$ **parallelly do**
3: $LB \leftarrow GenerateLowerBoundParallel(q, D)$
4: $realEDR \leftarrow InitialPriorityQueue()$
5: $candidateSet \leftarrow D$
6: **while** $candidateSet$ is not empty **do**
7: $simiTrajID \leftarrow ChooseSmall(LB, \eta)$
8: $realEDR \leftarrow CalEDRParallel(simiTrajID)$
9: $candidateSet \leftarrow FiltParallel(max(realEDR), candidateSet, LB)$
10: **end while**
11: $Result[q] \leftarrow realEDR.topID(k)$
12: **end for**

There is some challenges of performance in our proposed algorithm 1. In the most setting in actual situation, the number of cells are usually large, making the computation of lower bound a time-consuming process. For example, in Shanghai private cars dataset, an area of $45 \times 55 km^2$ is divided into 247500 cells, if threshold ϵ in EDR is set as $100m$. Besides, although through filtering, massive expensive EDR computations are still neccessary to execute. Attention that the computation cost of EDR between two trajectories is O(mn)(m is the length of trajectory 1 and n is the length of trajectory 2), an efficiency issue arises as the length of trajectories become longer. To overcome these challenges, we use GPU to accelerate lower bound generation and EDR calculation. However, this migration is not trivial because of the special architecture and programming model of GPU. Aiming to overcome these challenges and get better performance, we will introduce our strategies and implementation in detail at following part of this section.

1) Lower Bound Generation: The lower bound of EDR can be derived from cell-based trajectories. This process is proposed in [4]. We first generate frequency vector (FV) for each trajectory, storing the sample points the trajectory has in each cell. For example, in figure x, suppose the depth of GT-quadtree is 3. For a trajectory whose cell-based trajectory is [0,0,0,0,1,1,1,2,3,3], its frequency vector would be [4,3,1,2]. Frequency distance (FD), defined as the edit distance between two FVs, has close relationship with EDR between two corresponding trajectories. It is easy to find that insert, delete and replace operations in EDR is similar to adding one, subtracting one and subtracting one in one element then adding one in other element on FV correspondingly. But one adjacent case is special. If there is an replace operation between two adjacent elements of FV, indicating these two

corresponding cells are adjacent in grid, it may cause one unneccessary operation in EDR. For example, given two FVs, $v_1 =< 1,0,0,0 >$, $v_2 =< 0,1,0,0 >$, and corresponding trajectories $R_1 = (t_1, 0.9)$, $R_2 = (t_2, 1.1)$ and $\epsilon = 1$, replace operation is needed in FD but unneccessary in EDR. So, we can derive that FD is the lower bound of EDR.

Although the calculation cost of FD is lower than EDR, it still spends lots of time. There are two reasons. One is there are lots of trajectories in storage and for each of them an FD should be calculated. The other is if a small ϵ is set by user, the length of FV will be very long, causing even one FD is also computational cost. To accelerate this process, we distribute the FD computation workload of all queries to GPU evenly. To improve the throughtput of our query engine, FVs are pre-computed before the first query. When a query q comes, suppose there are N trajectories $t_1, t_2, ..., t_N$ in storage, N tasks computing FD between query trajectory and trajectories in storage are generated and each of them is assigned to a thread block, acting as the lower bound of EDR.

Algorithm 2 shows the procedure of generate FD in each thread block. Firstly we calculate the difference between two FVs (line 1-3) in parallel. Considering the adjacent case in edit distance calculation mentioned, we reduce the difference if two elements are neighbor cell in grid of GT-quadtree (line 4-12). Given an Morton encoded cell ID, its neighbors' ID can be generated in several cheap bitwise operations by thread of GPU. Finally, FD is calculated from the difference of two FVs by a parallel addition and comparison (line 13-17). With strategy that in each parallelly loop almost an equal number of elements are assigned to each thread in an sequencial increasing order, our algorithm achieves thread level load balancing and coalesce accessing pattern. It is worth noting that this process is also load balancing in task level because all FVs have the same number of elements.

2) Parallel EDR Calculatation: After generating lower bound and pruning, a mass of EDR calculations tasks are waited for executing. Each EDR calculation takes in two inputs trajectories $traj_1$ and $traj_2$, and output the EDR distance between them. For executing massive EDR calculations in parallel, the key is to finding an approach of dividing the each calculation process into independent sub-processes.

We designed an iterative framework for parallel EDR calculation. The EDR calculation, which is acutually a dynamic programming, can be divided into several independent steps of comparisons. Figure 4 shows an example of calculating the EDR between two trajectorie $traj_1$ and $traj_2$ with length m and n. Each value in a state (represented as $state[i][j]$) is calculated by comparing and choosing the minimun value of $\{state[i-1][j]+1, state[i][j-1]+1, state[i-1][j-1]+subcost\}$, according the definition of EDR in section 2. After all the states have been calculated, the $state[m][n]$ is the EDR between two trajectories. If we use slashes to group the states, we notice that the values of states in one slash rely on the value of states in two upper left slashes. For example, in figure 4, if we want to calculate the states on slash 3, the states on slash 1 and slash 2 are needed. Meanwhile, comparing

Algorithm 2 GenerateLowerBound

Input:
Frequency Vector of query trajectory $qFV = [qFV_1, qFV_2, ...]$; FV of the trajectory in task $tFV = [tFV_1, tFV_2, ...]$; Number of cells N_{cell};

Output:
Frequency Distance between qFV and tFV: FD

1: **for** each $i <= N_{cell}$ **parallelly do**
2: $dFV[i] \leftarrow qFV[i] - tFV[i]$
3: **end for**
4: **for** each $i <= N_{cell}$ **parallelly do**
5: **for** each j adjacent to i **do**
6: **if** the sign of $dFV[j]$ and $dFV[i]$ are different **then**
7: $x \leftarrow min(|dFV[i]|, |dFV[j]|)$
8: $dFV[i] \leftarrow dFV[i] + ((dFV[i] > 0) - 0.5) * 2$
9: $dFV[j] \leftarrow dFV[j] + ((dFV[j] > 0) - 0.5) * 2$
10: **end if**
11: **end for**
12: **end for**
13: **for** each $i <= N_{cell}$ **parallelly do**
14: $positive \leftarrow positive + (dFV[i] > 0) * dFV[i]$
15: $neagtive \leftarrow negative + (dFV[i] < 0) * dFV[i]$
16: **end for**
17: $FD \leftarrow max(positive, negative)$

operations within a slash have no relationship, inspiring us to invoke the calculations within the same slash in parallel, forming the basic idea of our solution.

Based on this framework, we propose a procedure of calculating EDRs in parallel on GPU. Given a set of EDR calculation tasks, we allocate each of them to a block to make GPU full-loading, in order to improve the throughtput. Also high-speed local memory allocated to each block permits us to handle different tasks on different blocks in parallel efficiently. In each block, the values of states are calculated in $(2 \times max\{m, n\} + 1)$ loops from $state[0][0]$ to $state[m][n]$. In each loop, all the values on a slash are calculated by mass of threads in parallel, with each state is assigned to one or more threads. For example, in figure 4, state on slash 1 is generated, and then slash 2, slash 3, ..., until the slash $(2m + 1)$.

In each loop, states in two before slashes are stored as shared variables because in CUDA programming model all shared variables will be stored in high speed local memory and calculation of states on slash i is related to states on slash $(i-1)$ and $(i-2)$. In this way all of states accessing transactions required from EDR calculation are from high speed local memory. In addition to this, we find that calculation of $state[i][j]$ needs for $traj_1[i]$ and $traj_2[j]$ and in some loops all of trajectory points are required. Also storing them as shared variables is surely a choice. However, for limited capabacity of local memory, this choice will impact the efficiency of GPU. This is because in GPU architecture, each SM can execute at most 8 blocks therotically at a time but all of shared variables of blocks are needed to be resident in local memory. So

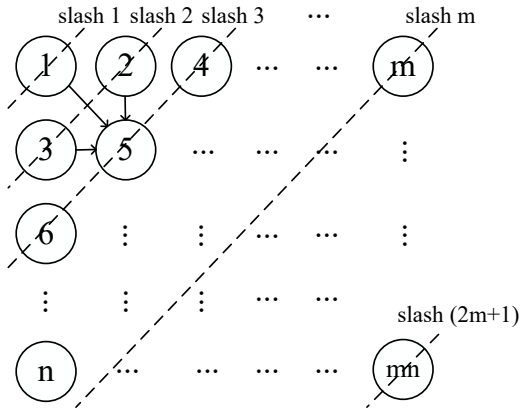


Fig. 4. An example of EDR calculation procedure

abusement of shared variables may limit the number of blocks running on an SM at the same time. For this reason, we choose

We store states according to the order of slash, as figure 4 shows. In this case, the data required by threads are near by, assuring the coalesce accessing pattern. We can see that the number of steps needed for EDRcalculation reduces from mn to $(2max\{m,n\}+1)$ comparing to not using GPU, and this reduction could be so obvious if trajectory length m or n is large, which is usually an actual situation because the sample interval of location device is short, e.g. 2s.

We propose an CUDA implementation of our strategy dealing with large-scale of EDR calculation from all top-k trajectory queries during a certain time slot. Each set of EDR calculation tasks from a query is handled by a single CUDA kernel. To keep GPU busy all the process, we use stream technology of CUDA to execute multiple kernels corresponding to different queries on GPU at the same time. Before these kernels running, trajectory data are loaded into GPU global memory first. Considering the low transferring speed between memory and GPU, we don't drop these trajectory data immediately after the kernel finishing because it may be reused by other kernels. However, the volume of the GPU global memory is usually so limited that not all trajectory data can be filled into it. To make an tradeoff, we maintain a memory pool on GPU storing trajectory data used by EDR calculation. If the pool is full, we use Least Recently Used (LRU) algorithm to drop some outdated trajectories, because there exists hotspot within queries. For example, trajectories in city center may be required by queries more frequently because the population in city center is denser than rural area. After the kernel finishing, it produce the EDR results for query handler to filter the candidates, as shown in line 9 in algorithm 1.

VI. Evaluation

In this section, we conduct a multiview experiment based on two real trajectory datasets to verify the performance of GTS. We first introduce the experimental environment, then evaluate the efficiency and scalability of index, and compare the performance of two kinds of queries with baseline and state-of-the-art works at last.

A. Environment

1) Dataset: We use two real life trajectories datasets to test the performance in different trajectories distribution. The first is Shanghai Private Car Data, containing xxx trajectorys of private cars in Shanghai collected from July, 2014 to April, 2015. The sampling rate... The size of the whole data is xxxGB. The second trajectory dataset is Geolife, the data collected by ....

In our experiment, we seem the sequence of sample points of a single car as a trajectory. If the time stamp of one point is over 30 minutes later than one before in a trajectory, we call this point a "gap". We split the trajectory into several new trajectories according to these gaps. This is because we usually only concern about the trajectory of a single route especially when handling similarity query, and trajectory with these gaps is usually not a meaningful single route. For example, a trajectory of a single car may include sample points from home to office and ones from office to home, and after spliting it two meaningful trajectories can be generated. After processing we get at total xxx.....

2) Experiment setup: In query performance part, we compare the query time consumption of two kinds of queries in different baselines and state-of-the-art systems. For range query, we randomly generate 1000 range queries with different area and reckon the time used after all of queries are finished. We implement three baselines: SharkDB,ST2I and GTS_{CPU}.xxxxxxxxxxx(the description of baselines) For similarity query, we first choose 1000 trajectories from dataset as the query set, and set different query parameters including k and ϵ . Then, we execute similarity query with this query set and reckon the time consumption.

We run all the experiments on a server equipped with ten-core Xeon E5-2650 v3 processor clocked at 2.3GHz, 64GB of RAM, 4TB of disk storage and an NVIDIA Tesla K80 GPU with 6GB graphical memory. Our system is implemented by C++ with CUDA 8.0, and operating system is CentOS 7.

Metric: throughtput and latency memory consumption

B. Index Evaluation

C. Query Performance

1) Range Query:

2) Top-k similarity Query: Compare with state-of-art method including TrajStore, SharkDB(Non-GPU method), ST2I+consistent storage(GPU method, intuitive approach, use s-t index to implement trajectory index) show that our system is more accurate than TKSimGPU(GPU method)

D. Impact of Parameters

E. Other Details Should Be Evaluated

VII. Related Work

A. Trajectory Storage and Indexing

B. GPU-accelerated Storage System

VIII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] W. Lee, W. Si, L. Chen, and M. Chen, "HTTP: a new framework for bus travel time prediction based on historical trajectories," in *SIGSPATIAL/GIS*. ACM, 2012, pp. 279–288.

[2] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire, "A gpu-based index to support interactive spatio-temporal queries over historical data," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, May 2016, pp. 1086–1097.

[3] Y. Zheng, "Trajectory data mining: An overview," *ACM TIST*, vol. 6, no. 3, pp. 29:1–29:41, 2015.

[4] L. Chen, M. T. Özsu, and V. Oria, "Robust and fast similarity search for moving object trajectories," in *SIGMOD Conference*. ACM, 2005, pp. 491–502.

[5] F. Lettich, S. Orlando, and C. Silvestri, "Processing streams of spatial k-nn queries and position updates on manycore gpus," in *SIGSPATIAL/GIS*. ACM, 2015, pp. 26:1–26:10.