An Efficient GPU-accelerated Framework for Big Trajectory Data Analytics

Bowen Zhang #, Yanmin Zhu #, Yanyan Shen #

# *Department of Computer Science and Engeneering, Shanghai Jiao Tong University*
Shanghai, China 200240
{zbw0046, yzhu, yy-shen}@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 analytics framework able to process mass of various kinds of queries efficiently. GPU, which has been widely equipped in data centers, can accelerate queries by handling them in parallel. However, existing GPU-accelerated trajectory storage systems are optimized for specific kind of query, suffering from the problem of efficiency when they are used for processing queries they are not optimized for. To solve this problem, we propose a framework optimized for the features of GPU which supports both two basic kinds of queries for big trajectory data. We design a unified storage component with an index called GT-quadtree with a cell-based trajectory storage, which combines and links quadtree, grid and trajectories together to support pruning methods for two basic kinds of queries. Based on the storage component, to make full use of the parallel power of GPU, we develop a query processing component optimized for the issues of GPU computing including load-balancing, coalesce memory accessing and less data transferring. We evaluate our system on two real-life trajectory datasets, which shows our system able to conduct two basic types of queries on large scale trajectory data efficiently. Moreover, our system achieves a speedup of 38x for range query and 67x for EDR measured top-k similarity query than implementing on CPU, demonstrating that our system really achieves the goal of accelerating both two kinds of queries on large-scale trajectory data by GPU.**

I. Introduction

As the number of mobile location devices (e.g., smartphones, cars, public bicycles, etc.) grows, increasing amount of spatial-temporal sequential data called trajectories are collected from the every corner of the world. For example, 327,474 trajectories are generated during July 1, 2013 and April 30, 2014 just from 9,144 private cars in Shanghai, which contains more than 247,494,133 sample points. There are a large amount of interesting and valuable information in these data to be exploited, such as route planing[1], trajectory pattern mining[2], travel time prediction[3] and so on. What plays an important role in big trajectory data analytics is trajectory query. Because thousands of queries are produced everyday in these applications, they call for a solution of efficient trajectory storage and retrieval framework. Most traditional databases and big data processing frameworks (e.g., MySQL, SparkSQL[4]) are designed for structured data, but they shows a poor performance when facing unstructured data such as trajectories.

Recent years various kinds of spatial indices and in-memory storage such as Simba[5] and SharkDB[6] are proposed to increase throughput and decrease latency of spatial queries for high-dimensional points. Most of them are designed for specific kind of query, e.g. range query or top-k similarity query. However, in many applications such as travel time prediction, both range query and top-k similarity query are required. In this situation, these approaches fail in the efficiency of executing queries which they are not optimized for. On the other hand, the performance of these approaches is limited by the single-core CPU. For example, it is reported in [7] that the execution time of a top-5 similarity query is about 70 seconds, which is quite slow for some interactive applications. Although multithread and multicore CPU technology mitigate this problem to some extent, but the speedup is limited to the number of cores on the CPU chip.

Having witnessed that GPU has been widely applied in some works[8][9][10] to accelerate spatial queries, it is an excellent idea to accelerate a large number of queries on big trajectory data by leveraging strong power of parallel computing. As a many-core architecture processor, GPU can achieve high throughput by executing tasks on thousands of cores at a time. However, as far as we know, only Zhang et.al. proposed a grid-based index[11] and then developed several algorithms about PIP-test[12] and similarity query[13] which leveraging GPU to accelerate the processing. But the similarity metric, which is called Hausdorff distance[14], used in this work is too simple to be practical in real life. In these years various popular similarity metrics which are based on global alignment are proposed, such as LCSS[15] DTW[16] and EDwP[7]. However, the computations of them are expensive because of the high complexity algorithm. It is highly valuable for trajectory analytics frameworks to accelerate these computations and reduce the time cost by some parallel computing devices such as GPU.

Inspired by these developments, we design and implement the GFBTA (GPU-accelerated Framework for Big Trajectory Analytics) framework, which can accelerate both range queries and top-k similarity queries with the metrics which are based on global alignment for historical trajectory data. It should be noted that in our work we take EDR as an example of metrics based on global alignment, and according to the work of Chen[17], it is possible to extend our work to other metrics such as DTW, ERP and LCSS in the future. In the framework, we design a unified storage component and a GPU-friendly query processing component optimized for the objectives of low query latency with acceptable memory occupation. Moreover, our design is friendly with GPU, which means we can make full use of parallel computing ability of it to accelerate query processing. As far as we have known, we are the first work which accelerates top-k similarity query with global alignments by leveraging GPU.

To support both two types of queries, we propose a novel trajectory storage component. We observe that the histogram approach[17] used in pruning unnecessary EDR calculation is similar to a fixed grid, which has been widely used as the index for GPU-accelerated range query. On the basic of this observation, we subtly adopt the characters of Morton-based encoding and PR-quadtree[18] to design an index called GT-quadtree, which can be helpful for reducing the computational cost for both range query and top-k similarity query. We apply Morton-based encoding to the cells' identifies in GT-quadtree and store points within a cell continuously in memory to reach the goal of achieving a coalesce access pattern when they are needed to copied to GPU, which is a requirement of efficient GPU programming. Moreover, by importing Morton-based encoding to GT-quadtree it also benefits the load balancing and the tradeoff between the elapsed time in filter phase and refinement phase in query processing.

To make full use of parallel computing ability of GPU, we design a special query processing component supporting two basic kinds of queries. For range query, we migrate the query algorithm used for quadtree index to GT-quadtree, which is load balancing and has the coalesce accessing pattern to avoid the performance loss. For top-k similarity query, we leverage the potential independence of the procedure to design a scalable task division strategy for a batch of calculation of EDR. Otherwise, we design a special placement organization of elements for the state matrix stored in GPU global memory, which satisfies coalesce accessing pattern when calculating EDR on GPU. We also maintain a buffer table for two types of queries respectively to reduce low-speed data transferring on PCI-E interface by avoiding duplicated data transferring.

The contributions of our work can be concluded as follows:

- We propose a GPU-accelerated framework for big trajectory data analytics which supports both range query and top-k similarity query with popular EDR-like(metric whose procedure is similar to EDR?) metric.
- We design an trajectory storage component with a quadtree-grid hybrid index over large-scale trajectory data, which supports indexing for both two basic kinds of queries.
- We propose a novel solution of accelerating the calculation of EDR distance based on the trajectory storage component by exploiting GPU's parallel computation ability.
- We implement our system on two real-life datasets and evaluate its performance, proving that it gets about 38x speedup for range query and 67x speedup for top-k similarity query respectively comparing to the implementation on single-core CPU with an acceptable memory occupation.

The rest of paper is organized as follows. In Section 2 we introduce some background of our work. After that, in section 3 we briefly introduce the architecture of our framework. We propose our design of storage component in section 4. The query processing component is described in detail in section 5. In section 6, we evaluate our system and results are reported. Some related works are summarized in Section 7, while Section 8 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. Commonly-used notations are listed in table I.

TABLE I
COMMONLY-USED NOTATIONS

Notation	Description
p	a sample point with latitude, longitude and timestamp
t	a trajectory, the sequence of some sample points
MBR	a rectangle range in geography
$\mathbb{D}$	a trajectory dataset
$RQ(MBR, \mathbb{D})$	points in trajectories of $\mathbb{D}$ and within MBR
$EDR(t_1, t_2)$	the EDR between two trajectories, t_1 and t_2
$TSQ(t_q, k, \mathbb{D})$	k most similar trajectories of t_q in $\mathbb{D}$
$dist(p_1, p_2)$	euclidean distance between p_1 and p_2

Definition 1 (sample point): A sample point of trajectory $p = (x, y, time)$ is a three-dimensional data which include spatial information represented by (x, y) and time stamp $time$. For simplicity, we assume that all coordinates of sample points have been transformed into 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. Meanwhile, we say that $p_i \in t$ if p_i is a sample point of trajectory t. 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 dataset $\mathbb{D}$ including trajectories $\{t_1, t_2, \ldots, t_{|\mathbb{D}|}\}$, our goal is answering the range query and top-k similarity query. Here we formulate these two kinds of query. The range query takes an Minimum Bounding Rectangle (MBR) as a condition, then retrieves the sample points from trajectories whose coordinates are within the MBR. Here we formulate the range query as follow.

Definition 3 (Range Query): Given the the MBR of range MBR and the trajectory dataset $\mathbb{D}$, the range query $RQ(MBR, \mathbb{D})$ is defined as:

$$RQ(MBR, \mathbb{D}) = \{p | \exists t \in \mathbb{D} \ s.t. \ p \in t, p \in MBR\} \quad (1)$$

Top-k 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. We first define EDR[17], the metric of similarity between trajectories used in our work.

Definition 4 (EDR): 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 between them is calculated by:

$$EDR(t_1, t_2) = \begin{cases} n & \text{if } m = 0 \\ m & \text{if } n = 0 \\ \min\{EDR(Rest(t_1), \\ Rest(t_2) + subcost), \\ EDR(Rest(t_1), t_2) + 1, \\ EDR(t_1, Rest(t_2)) + 1\} & \text{otherwise} \end{cases} \quad (2)$$

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

Based on the definition of EDR, we define the problem of top-k similarity query as follow.

Definition 5 (Top-k Similarity Query): Given a trajectory t_q, the k value and the trajectory dataset $\mathbb{D}$, the top-k similarity query $TSQ(t_q, k, \mathbb{D})$ is defined as:

$$\begin{aligned} TSQ(t_q, k, \mathbb{D}) = & T_s \subseteq \mathbb{D} \ s.t. \ \forall t_1 \in \mathbb{D}/T_s, \forall t_2 \in T_s, \\ & EDR(t_q, t_2) \leq EDR(t_q, t_1), |T_s| = k \end{aligned} \quad (3)$$

B. GPU computing using CUDA[19]

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, and threads in different blocks can use a slower GPU global memory to exchange information. This requires us to divide the task properly into different blocks properly.

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 read/write data from global memory, if the data each thread needs are nearby, GPU will coalesce hundreds of small accessing request to several big continuous accessing transactions. Because GPU has a high bandwidth but high latency global memory, by this coalescing threads can get needed data in a pretty small number of clock cycles. For example, if thread 1,2,..,32 access the data at offset 33,34,...,64 respectively, 32 data loading requests will be combined to one single transition. So if program access data from global memory, it is optimal when data read/writen by all threads in a block are stored continuously, and ensuring an optimal memory access pattern is significant for making full use of GPU. 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, there is a tradeoff about the layout of large-scale trajectory data. The capability of GPU global memory is limit, for example, 12GB in most of ultimate level GPU devices. So we may think about store all data in host memory. However, PCI-E interface, the bridge between host memory and GPU, has much lower bandwidth than global memory which is inside GPU. This requires usto reduce data transferring between main memory and GPU as much as possible to avoid the performance loss. Comparing to other parallel computing tools based on multi-core CPU such as MPI, these three issues raise a challenge for GTS to get a high speedup on GPU.

III. System Architecture

A. Overview

Our system includes two parts, storage component and query processing component. As the Figure 1 shows, raw trajectories are preprocessed and divided 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 processed in query processing component, with the acceleration of GPU, outputing query results. It's worth noting that our system is designed for query-only applications, and we do not concern about the manipulation of data after building index. We then introduce these two components briefly.

B. Storage Component

In the design of storage component, we exploit the potential oppotunities of quadtree with Morton encoding[18] with

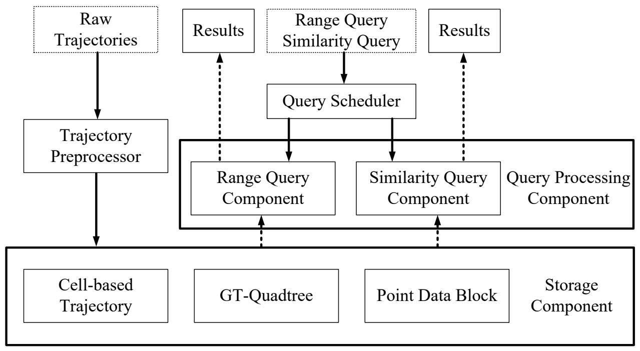


Fig. 1. System Architecture

the consideration of GPU and propose an index called GT-quadtree. 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. All data are stored in host memory, because in our strategies it is CPU that executes pruning process. 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 two basic kinds in parallel, optimizing for executing time per query. Different kinds of queries are executed in different query components, based on the same storage component. In both two query components, we use GT-quadtree index to filter the candidates, then accelerate refinement by leveraging GPU. 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 GPU computing mentioned before. At last, we propose the construction of our index.

A. Index and Storage Design

Index performs as an important role in storage system by pruning unnecessary 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 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[17], 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.

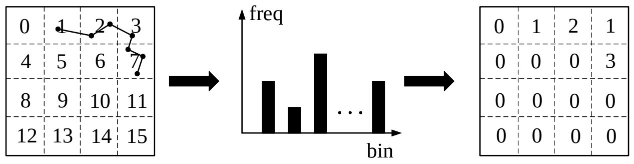


Fig. 2. Example of the trajectory in area (left), the histogram of this trajectory (middle) and the grid that can represent the histogram's value (right)

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 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.

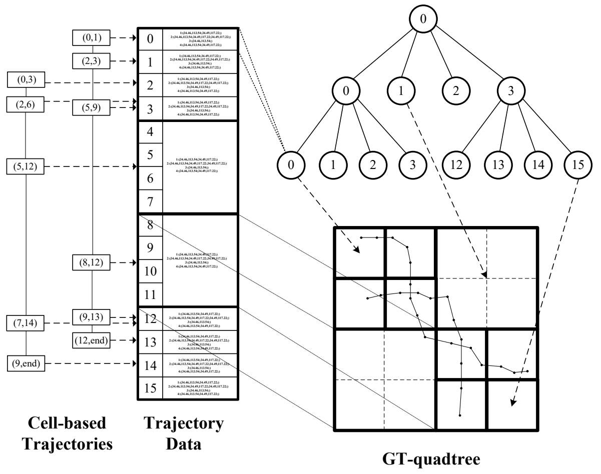
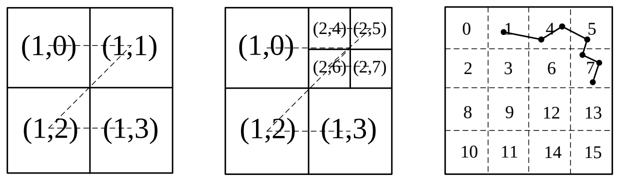


Fig. 3. Example of the trajectory in area (left), the histogram of this trajectory (middle) and the grid that can represent the histogram's value (right)

We adopt the idea of PR-quadtree[18] 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 3 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 M_{cell}. Also, if the number of points in a node is less than M_{cell}, 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 4 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 4, the parent of node $(2, 4)$ in figure 4(b) is $(1, 1)$ in figure 4(a). This process is done by profroming a two-bits right shift on $100(4)$, getting $1(1)$. This feature of GT-quadtree benefits both range query and similarity query, which will be illustrated in Section V.



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

Fig. 4. 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 global alignment. 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 4, 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. Index Construction

(This part differ very small to quadtree building)

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 4(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 [18], we build a new quadtree as our final index, in which the number of each cell not exceeding a given threshold M_{cell}, 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 M_{cell} will be divided into four quadrants, forming the children nodes in next level, meanwhile other nodes are labeled as leaf nodes, containing less than M_{cell} points. This procedure will be invoked recursively until there is no node to be divided. For example, in figure 4, node (1,1) in 4(a)is divided into (2,4), (2,5), (2,6) and (2,7) in 4(b).

1) Complexity Analysis:

V. GPU-accelerated Query Processing

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

There have been a solution[11] leveraging GPU to handle range query based on quadtree. However, this solution is based on a fact that all the points will be transfered into GPU memory, which is not realistic at a big data scene. In our solution, by integrate a usage table technique, we can also achieve a similar speedup even if data are not all in GPU memory.

Similar to the most of related work, 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.

1) filter phase: 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, the trajectory data within retrieved node are needed to be in GPU global memory. It is low efficient if duplicated data are transfered from host to GPU global memory, so we maintain a table recording the nodes whose data remain in GPU global memory and its corresponding pointer. Based on this technique, we first check whether the data of node are in GPU global memory. Only if neccessary, we transfer data of this node to GPU, otherwise the pointers of data are directly used to locate the required data. It is worth noting that CPU continue traversing the GT-quadtree simultaneously when transfering data, which can hide the latency caused by low-speed PCI to some extent.

2) refine phase: 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. To avoid the bottleneck due to the pairs which have larger number of points to be verified than others, we divided these pairs into several pairs containing no more than M_{cell} points. In each thread block 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 the quantities of points within nodes in GT-quadtree are all nearly M_{cell} points, the load balance is achieved in this step. Refinement procedure on multiple GPUs is similar, so we will not introduce it detailly because of the space limitation.

3) Complexity Analysis: We analyse the complexity of our approach of top-k similarity query.

$$Cost_{RQ(GPU)} = \sum_{q=1}^{N_Q} \frac{\max\{\rho L_{cell}^2, M_{cell}, N_{core}\}}{N_{core}} \frac{\frac{\rho S_q}{M_{cell}}}{N_{SM}} \quad (4)$$

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. [7][] Our work can be migrated to other measurements whose calculation is based on dynamic programming algorithm, such as LCSS[15]. However, considering that GPU efficiency is low if the task load is small, it is suboptimal to directly migrate algorithm proposed by [17] to GPU, because this algorithm execute only one EDR calculation after a filter phase. As a solution, we adapt this algorithm to fit with GPU architecture and design a new 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 unnecessary 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.

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 $45 \times 55km^2$ 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 EDR calculation with GT-quadtree. However, this migration is

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 $k$,$\eta$;
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
```

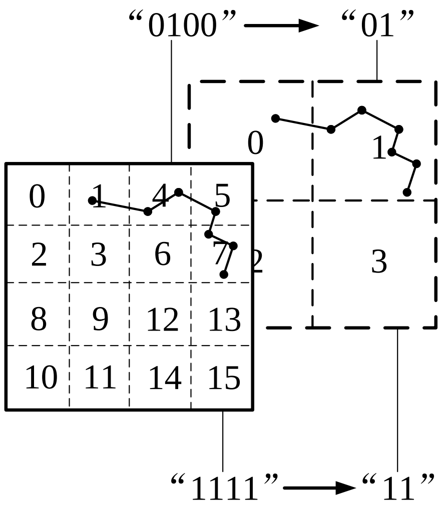


Fig. 5. An example of virtual grid with 4 cells and the transformation between IDs in different grids

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 [17]. We first generate frequency vector (FV) for each trajectory, storing the sample points the trajectory has in each cell. For example, in figure 5, suppose the depth of GT-quadtree is 3. For a trajectory whose cell-based trajectory is [1,4,4,5,7,7,7], its frequency vector would be [0,1,0,0,2,1,0,3,0,...] of length 16. 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 unnecessary 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 unnecessary 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. Aiming to these two issues, we proposed the strategies respectively. Firstly, widely used multicore CPU allows us to distribute the FD computation workload of all queries to different cores evenly, and then the calculation of all FD can finish in shorter time because the tasks are divided. Secondly, for the FD computation in each query, inspired by a solution proposed in [17], we build a virtual grid, in which the length of cell is larger than that of GT-quadtree's fixed grid, to reduce the length of FVs. It is trivial to execute FD calculation of different queries by using multithreading technology, so we then introduce the virtual grid in detail.

The virtual grid can be seen as all of nodes in one layer of GT-quadtree. Figure 5 is an example: a virtual grid with 4 cells is built based on a fixed grid with 16 cells. Chen[17] has proved that FD is still the lower bound of EDR as long as the length of cell in grid is the multiple of ϵ, no matter how large it is. So, if we use the virtual grid to calculate FD, the cost of computation will decrease. Owing to the Morton encoding of GT-quadtree, the virtual grid can be used without the requirement of extra memory consumption, because we can directly get which cell in the virtual grid contains a given point and then generate FVs. For example, in figure 5, the original FV of the trajectory can be simplified into [1,5,0,0], whose length is decreased from 16 to 4, just by a transformation implemented by right shifting. However, it should also be pointed out that a tradeoff should be found between too large and too small cells because the larger cells are, the looser bound is. We set a parameter called VC permitting the configuration of this tradeoff, which indicates how many right shifting operations are performed to get the transformation from original FV to simplified one in virtual grid. In above example, the VC parameter is 2.

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 cal-

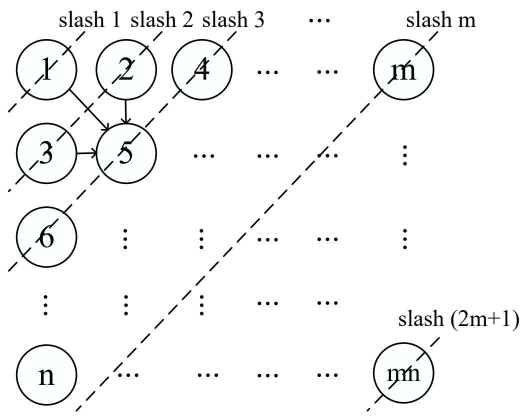


Fig. 6. An example of EDR calculation procedure

culation. The EDR calculation, which is acutually a dynamic programming, can be divided into several independent steps of comparisons. Figure 6 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 6, if we want to calculate the states on slash 3, the states on slash 1 and slash 2 are needed. Meanwhile, comparing 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, as algorithm 2 shows, whose implement is based on CUDA. Given a set of EDR calculation tasks, we first assign each of them to a block to make GPU full-loading, in order to improve the throughtput(line 1-2). In each block, the values of states are calculated in $(2 \times max\{m,n\}+1)$ loops from $state[0][0]$ to $state[m][n]$ slash by slash (line 10). For example, in figure 6, state on slash 1 is generated, and then slash 2, slash 3, ..., until the slash $(2m+1)$. In each loop data on two before slashes are stored in high-speed local memory of each SM, which is allocated before starting loops (line 6), because they will be frequently accessed by threads. Noting that the length of trajectory is almost smaller than 2048, it is enough for space of local memory to store these data. As for the state matrix of this calculation, we store them according to the order of slash, as figure 6 shows. In this case, the data required by threads are nearby, assuring the coalesce accessing pattern.

In each thread block, based on the data of previous two slashes, all the values on current slash are calculated by mass of threads in parallel by equation (1) mentioned in Section II (line 11-16). After finishing the calculation on current slash i, the states on slash i-2 stored in local memory are flushed into state matrix in global memory, and then states on slash i-1 and i become the new "two last slashes" (line 17-22). The result of EDR calculation of this block can be extracted from state matrix after all loops (line 25).

We can see that in our solution, the number of steps needed for EDR?calculation 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.

Algorithm 2 Parallel EDR Calculation

```
Input:
    Two sets of trajectories: $T1, T2$
    Threshold of EDR: $\epsilon$
    Parallel parameters: $blockNum$, $threadNum$
Output:
    all EDR distance between trajectories in two sets:
    $EDR(t_1,t_2), \forall(t_1,t_2) \in T1 \times T2$
 1: assign each $(t_1,t_2)$ pair to a thread block
 2: initial an array to store results of all pairs assigned:
    $result[blockNum]$
 3: for each block $bID$ parallelly do
 4:     initial a matrix in global memory to store all states:
        $state[len(t_1)+1][len(t_2)+1]$
 5:     $maxLength \leftarrow max(len(t_1), len(t_2))$
 6:     initial a matrix in local memory to store states in last
        two slashes: $lastTwoSlash[2][maxLength+1]$
 7:     $lastTwoSlash[0][0] = 0$
 8:     $slashNum \leftarrow len(t_1) + len(t_2) - 1$
 9:     $loopPerThread \leftarrow \lceil maxLength/threadNum \rceil$
10:     for each slash $i$, $i \in [1, slashNum]$ do
11:         for each thread $tID$ parallelly do
12:             initial an array in GPU SM's register to store states
                calculated: $tempState[loopPerThread]$
13:             for each loop $j$, $j \in [0, loopPerThread-1]$ do
14:                 update $tempState[j*threadNum+tID]$ using
                    states on last two slashes
15:             end for
16:         end for
17:         for each thread $tID$ parallelly do
18:             for each loop $j$, $j \in [0, loopPerThread-1]$ do
19:                 flush $lastTwoSlash[0][j*threadNum+tID]$
                    to $state$ in corresponding position
20:                 $lastTwoSlash[0][j*threadNum+tID] \leftarrow$
                    $lastTwoSlash[1][j*threadNum+tID]$
21:                 update $lastTwoSlash[1][j*threadNum+tID]$
                    using $tempState[j]$
22:             end for
23:         end for
24:     end for
25:     $result[bID] \leftarrow state[len(t_1)][len(t_2)]$
26: end for
```

We propose a multi-GPU implementation of our strategy dealing with large-scale of EDR calculation from all top-k trajectory queries by dividing the set of EDR calculation tasks into several equal size part, to achieve a load balancing and maximum usage of multiple GPUs. Before running these tasks, we use an independent memory allocated table (MAT) to store trajectory data when they are loaded into GPU global memory first, because in this way if a trajectory has been loaded into global memory, by looking for MAT we can avoid the duplicated low-speed data transfering between memory and GPU. 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 and integrate it to MAT. If the pool is full, we use Least Recently Used (LRU) algorithm to drop some outdated trajectories. This strategy is efficient in real life 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, EDR calculation results from different GPUs are collected and query engine then filters the candidates, as shown in line 9 in algorithm 1.

3) Complexity Analysis: We analyse the complexity of our approach of top-k similarity query.

$$Cost_{EDR(CPU)} = \sum_{q=1}^{N_Q} len(t_q) * len(t_D) \tag{5}$$

$$Cost_{EDR(GPU)} = \sum_{q=1}^{N_Q} \frac{\lceil \frac{len(t_q)}{N_{core}} \rceil + \lceil \frac{len(t_D)}{N_{core}} \rceil}{N_{SM}} [len(t_q)+len(t_D)] \tag{6}$$

VI. Experiment

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. Experiment Setup

1) Dataset: We use two real life trajectory datasets which are collected from two largest cities of China respectively to test the performance in different trajectories distribution.

SHCAR SHCAR is the trajectories of some private cars in Shanghai City, which contains 327,474 trajectories and 75,188,293 sample points. Some extra information such as CARID, direction and OBD information are also included in this dataset. All the data were collected from July, 2014 to April, 2015. The sampling rate is about 10 seconds. The size of the raw data file is 8.96GB.

GeoLife The GeoLife dataset is published by Microsoft Research Asia, which contains the trajectories from 182 users in a period from April 2007 to August 2012[20][21][22]. We pick all the sample points within Beijing City, including 30,325 trajectories and 19,143,208 sample points. The size of raw data file is 2.11GB.

TABLE II
Parameters ranges and default values

Parameter	Meaning	Range	Default
L_{cell}	size of each cell in grid	0.05 - 0.4	0.1
M_{cell}	max num. of points in a cell	256 - 4096	1024
VC	the VC parameter of virtual grid	1 - 50	15
S_{RQ}	area of range query's MBR	0.01 - 4	0.5
k	k value of top-**k** similarity query	8 - 128	32
N_Q	num. of queries	256 - 8192	1024
L_{QT}	length of query trajectory	256 - 8192	1024

2) Data Preprocessing: In our experiment, we seem the sequence of sample points of a the same car as a trajectory. If the difference between two time stamps of consecutive points is larger than 30 minutes in a trajectory, we call this point a "gap". We then 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 preprocessing we get at total 327,474 trajectories.

3) Parameters: To systematically test the performance of our system, we conduct our experiments under various parameter settings. Table II shows the range and default value of parameters we test in experiments. For range query, there are four parameters affecting the performance. In real life, queries from users vary from area, so we test for different size of MBB in range query. We also test the situations of different number of queries to evaluate the scalability of GTS. For top-k similarity search, the execution time under different k value and number of queries are recorded. As we show in section V the length of trajectory determines the complexity of EDR calculations so we test for different length to see whether GTS gets high performance at different situation. Size of cells are altered in all of experiments of both two kinds of queries.

4) Baselines: For range query, we implement two state-of-the-art systems supporting range query on GPU: STIG[8] and FSG[11]. However, these two systems are designed facing to spatial-temporal points rather than trajectories, so we add a data field in each point to represent the trajectory ID of it. To show the acceleration performance, we also implement our approach on multicore CPU as the baseline method.

STIG This approach use kd-tree as the basic data structure, in which each leaf corresponds to a block rather than a point. We choose in-memory CPU-GPU hybrid strategy of it, which means when executing range query, kd-tree is firstly traversed by CPU and some candidate blocks are returned, then each block is refined by a block in GPU. We set the parameter of block size to 20000, same as default M_{cell} in our approach.

FSG This approach leverage a flat grid-file based indexing to accelerate point-in-polygon queries for analysing taxi trip data. Similar to our approach, it firstly find cells which overlap the

MBR of queries, then generate cell-polygon pairs and send them to GPU. In next phase each pair is processed by an SM, achieving the goal of acceleration. We set the parameter of size of cell to 0.02, same as default L_{cell} in our approach. R value is set to 0.0025.

GTS-MCPU The CPU version of GTS. In this approach each $\langle node, query \rangle$ pair is assigned to a thread of CPU rather than an SM of GPU to achieve parallelism. All the parameters are set the same as which in GTS.

For similarity query, we only implement the original EDR-based top-k similarity query algorithm proposed in [17] on multicore CPU as the baseline because as far as we know we are first to utilize GPU to accelerate EDR based top-k similarity query.

EDR-MCPU We modify original EDR-based top-k similarity query method to a multithread version by assigning each query task to a thread. The parameters of this method are the same as which in our approach.

5) Experiment Overview: ???All other parameters are tuned to the optimal case.

After that, the scalability is tested. As there is no method to shut some cores in GPU, we can only test the situation with different number of GPUs.

In query performance part, same as the most of previous works, we use query latency as our metric. We compare the query time latency of two kinds of queries in different baselines and state-of-the-art systems in a large query set situation. When testing range query, we randomly generate range queries with different areas and positions and reckon the time consumption during finishing all of queries. To reflect the true working environment as much as possible, the chosen positions are restricted to the central district of Shanghai ($31.11°N - 31.36°N, 121.39°E - 121.58°E$). For similarity query, some trajectories are selected randomly from dataset as the query set. During the query, for each trajectory in query set, the result of top-k similarity query is returned under the settings of different query parameters including k and ϵ. Queries are handled with formed query set and time consumption in both baseline method and GTS are then calculated.

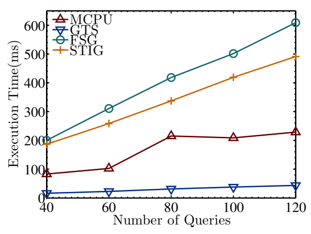
We run all the experiments on a server equipped with two ten-core Xeon E5-2650 v3 processor clocked at 2.3GHz, 64GB of RAM, 4TB of disk storage and an NVIDIA Tesla K80 GPU with 6GB graphical memory. Our system is implemented by C++ with CUDA 8.0, and operating system is CentOS 7.

B. Query Latency

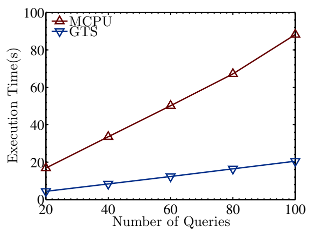
In this section we evaluate the performance of range query and top-k similarity query under SHCAR dataset. For range query, we compare our approach with three baselines described in Section VI-A. For top-k similarity query, we only compare our system with the multicore-CPU implementation of original EDR based top-k similarity query because there is no other GPU-accelerated query processing approach based on EDR.

Range Query We evaluate the efficiency of three baselines and our system under different workloads. The workload is controlled by the number of range queries with the same MBR. The result is shown in figure7, we can see that our approach outperforms other three baselines. The reason that two GPU baseline, FSG and STIG, perform worse that MCPU is because the main cost of them is the data transfering between GPU and CPU, which is getting larger as the increasing scale of queries. In our system, owing to the memory allocation table, we can avoid duplicated data transfering on the slow PCI-E interface. On the other hand, in our approach refinement workload on GPU is even, leading to more efficient parallelism. We can also see that query execution time of our approach grows linearly with increasing number of range queries. Also hundreds of queries can be finished within 200 ms, which meet the requirement of real-time service.

Top-k Similarity Query We test the performance of the baseline method and our system. The number of queries are set between 10 and 70 to show the time consumption under different workload. From the result shown in figure7, we can see that GPU-based approach we proposed outperforms original method implemented on multicore-CPU under all workload, because of much higher throughtput of GPU. Moreover, as the increasing number of queries, the gap between these two methods becomes larger. This is because our method mainly accelerate the calculation of EDR distance, and the propotion of time consumed in this part under high workload is much larger than that under low workload. Note that the time taken to finish top-5 similarity query for 70 trajectories having 1024 points in our GPU-based implementation is only about 38s, which is drastically shorter than that of implementation on multicore-CPU.

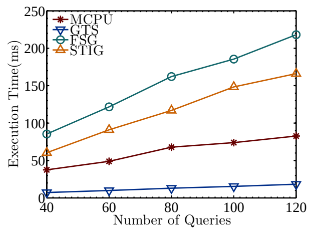


(a) Execution time for different number of range queries

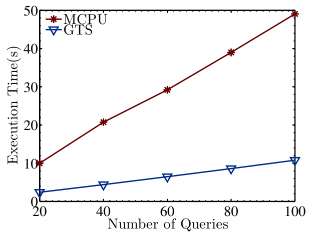


(b) Execution time for different number of top-k similarity queries

Fig. 7. Query performance for range query and top-k similarity query under different workload

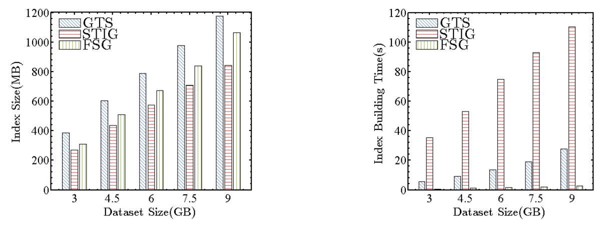


(a) Execution time for different number of range queries



(b) Execution time for different number of top-k similarity queries

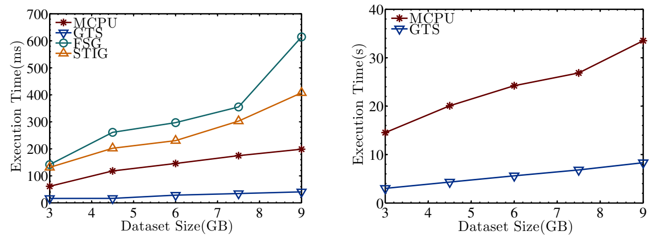
Fig. 8. Query performance for range query and top-k similarity query under different workload



(a) The memory occupation of index on different size of dataset

(b) The index building time of index on different size of dataset

Fig. 9. The memory occupation and building time of index under different size of dataset



(a) Execution time of range queries under different size of data

(b) Execution time of top-k similarity queries under different size of data

Fig. 10. Execution time of 80 range queries (left) and 40 top-k similarity queries (right) under different size of dataset

C. Indexing Cost

We then test the indexing cost of our system. We measure the memory occupation and building time of different indices in the baselines and show it in figure 9. All the parameters are set to default. We can see in the evaluation of 9GB dataset our approach only use about 80MB more memory to support efficient top-k similarity query. This is because some information such as fixed grid and cell-based trajectories can be reused in both range query and top-k similarity query, so only a quadtree causes the extra occupation of memory. We argue that it is worthy to pay this small cost as a unified index for both two kinds of queries because in the most of applications we concern more about speed of query processing than a little more memory occupation. From figure 9, we can see our approach spends less time in building index than STIG, because in STIG there is a large amount of computation of finding medium value when generating a layer of kd-tree.

D. Speedup Ratio

In this section we study the speedup attained by our GPU-based approach under default parameters with one and two GPUs are used. Note that 2496 CUDA cores whose frequency is 562MHz are avaliable in each GPU. We compute the speedup against the implementation on a single-core CPU whose frequency is 2.3GHz. This study explains two benefits. First, it shows that our GPU-based implementation can outperform traditional implementation on CPU. Second, it demonstrates that a near-linear improvement of efficiency can be achieved by adding more GPU devices.

TABLE III
SPEEDUP ACHIEVED IN TWO DATASETS

Dataset	**SHCAR**		**GeoLife**	
# GPU	**1GPU**	**2GPU**	**1GPU**	**2GPU**
Range Query	20.07	37.63	18.03	32.43
Top-k Similarity Query	35.54	68.53	38.94	74.95

Table III shows the speedup ratio for two kinds of queries. Our approach achieves about 80x speedup in evaluation of top-k similarity query for single GPU, and 42x in dual GPUs environment. The high speedup ratio comes from the parallel execution of compute-intensive calculation of EDR distance. For range query, although our approach achieves about 8.28x speedup for single GPU and 14.59x speedup for dual GPUs, slow data transfering between host memory and GPU may restrict the speedup ratio. This is because only some comparison operations are needed in the refinement procedure of range query, which means it is not a compute-intensive task.

E. Scalability

We investigate the scalability of all of the approaches in this part. Figure x presents the results under different size of data in the range from 3GB to 9GB.

Range Query From figure x(a) we can see that in the evaluation about range query, all the approaches increases almost linearly when we enlarge the MBR of the query. This is determined by the fact that more candidates are included in range queries if there are more points in dataset. We can also see our approach shows a better performance than other baselines in the test of each size of dataset, proving our system has a good scalability on range query.

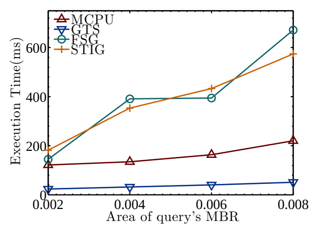
Top-k Similarity Query The execution time of top-k similarity queries on different size of dataset is shown in figure 10. We can see as the growing volume of data, both two approaches consume more time on queries and the benefit of acceleration by using GPU becomes more obvious. It also proves that our system works excellently on large-scale dataset.

F. Parameters Tuning

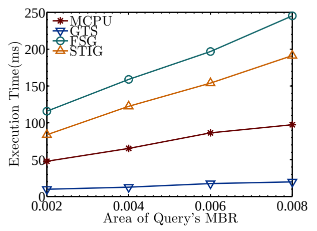
In this section we study the effects of parameters in GTS. Here are totally seven parameters in our experiment as TableII shows. Some of them can be categorized as query parameters, which are properties of queries and can be evaluated in both other systems and GTS, such as the area of MBR and k. Others are system parameters, which only exist in our system. For this reason, we show results of all baselines when evaluating the execution time in different query conditions to demonstrate both the expected performance in various environment and the effects of these parameters. Meanwhile, system parameters are evaluated under different query scales to show the effects, which are only based on our approach.

Range Query There are three parameters about range query. We will evaluate the effects of them and explain the reasons.

S_{RQ} is the only one query parameter about range query. Figure11 shows the time consumption of queries with different S_{RQ}. We can see in all approaches, the time consumption rises as the area of MBR increases from 0.002 to 0.008. We can see

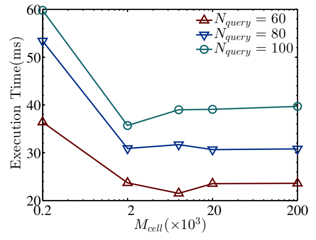


(a) Execution time of 80 range queries under different area of MBR

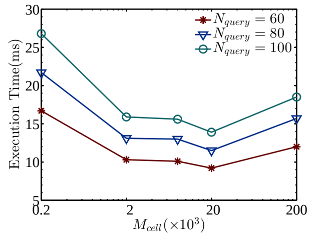


(b) Execution time of 80 range queries under different area of MBR

Fig. 11. The execution time (left) and speedup ratio achieved (right) in the top-k similarity queries

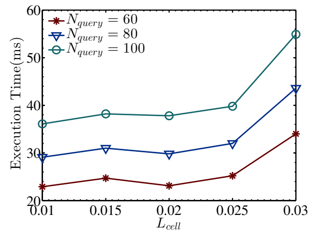


(a) Execution time of 80 range queries under different M_{cell}

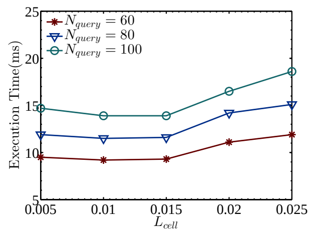


(b) Execution time of 80 range queries under different M_{cell}

Fig. 12. The execution time (left) and speedup ratio achieved (right) in the top-k similarity queries



(a) Execution time of 80 range queries under different L_{cell}



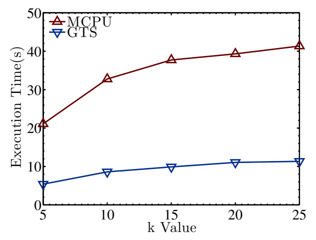
(b) Execution time of 80 range queries under different L_{cell}

Fig. 13. The execution time (left) and speedup ratio achieved (right) in the top-k similarity queries

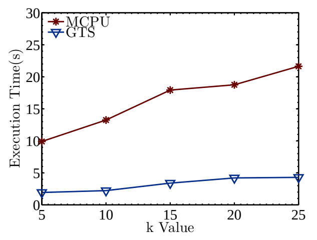
that this trend is nearly linear, because the number of nodes overlapped by the MBR is basically propotional to the area of MBR.

There are two system parameters about range query: L_{cell} and M_{cell}. As shown in figure **??**, under $M_{cell} = 2000$ and $M_{cell} = 20000$ our approach achieves the best performance in all tests. We can see a downward trend of execution time as decreasing M_{cell}. This is because if M_{cell} is too small, each SM of GPU need only finish several tests of whether a point is in MBR, leading to not fully usage of GPU. As a consequence, time consumption other than refinement itself such as memory allocation, data copy and kernel launching becomes the main source of cost. However, as M_{cell} continues to increase, the workload of some refinement tasks are larger than others, causing a slight increasing of the execution time.

Figure 13 shows the running time by varying the number of L_{cell} from 0.01 to 0.03. From the results, we observed that the running time tends to get larger under big L_{cell}. The reason is that in this situation, a larger quantity of points are collected as the candidates in filter phase. To explain it,

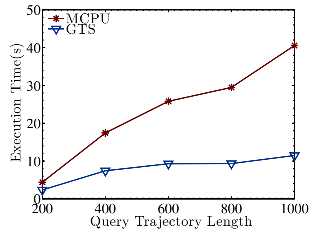


(a) Execution time of 40 top-k similarity queries with different query trajectory length

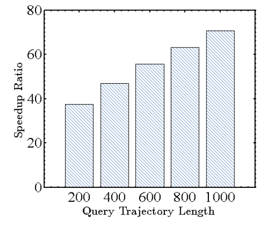


(b) Speedup ratio of 40 top-k similarity queries with different query trajectory length

Fig. 14. The execution time (left) and speedup ratio achieved (right) in the top-k similarity queries



(a) Execution time of 40 top-k similarity queries with different query trajectory length



(b) Speedup ratio of 40 top-k similarity queries with different query trajectory length

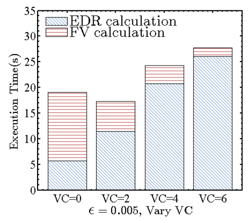
Fig. 15. The execution time (left) and speedup ratio achieved (right) in the top-k similarity queries

considering an extreme situation that L_{cell} is so large that there is only one cell in the whole plane.In this situation all of points are verified in refinement phase, which means that the whole process degrades to sequencial scan. However, this does not mean that smaller L_{cell} is always better, because larger number of cells are generated, resulting in a slight increasing performance but much higher memory cost.

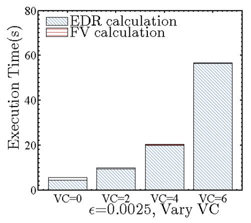
Top-k Similarity Query In this section, we study the effects of parameters in top-k similarity query, including L_{query}, k and VC. We omit ϵ because it is considered when studying the effect of VC.

Figure x shows the execution time of our approaches for different value of k. We can see that for all k values our approach achieves a better performance than CPU-based implementation, and the execution time goes higher as the k value increases. This is because a larger k means less trajectories are filtered than that of small k. For example, for $k = 5$, trajectories whose lowerbound of EDR are higher than the 5-th highest EDR in the priority queue will be filtered, but if $k = 25$ less trajectories will be filtered because more trajectories will have a lowerbound which is lower than the highest EDR of trajectory on the tail of the priority queue.

We then study the effects of the length of trajectories in query set. To concern on the effects of trajectories' length, we control other factors by using trajectories with the same cell-based trajectory but different length to form a query set. Figure x shows the execution time and speedup ratio for query sets which are built with trajectories of different length. It can be seen that speedup ratio rises linearly as the growing length of trajectories in query set. This phenomenon proves the cost computed in equation (x), meaning that our approach



(a) Execution time of 40 top-k similarity queries with different query trajectory length



(b) Speedup ratio of 40 top-k similarity queries with different query trajectory length

Fig. 16. The execution time (left) and speedup ratio achieved (right) in the top-k similarity queries

can achieve a higher speedup if trajectories in query set are longer.

VC is a system parameter which has effects on the process of pruning, as we described in section V. Because for different ϵ the most suitable VC value is not the same, we study the effects of VC based on the situation that ϵ equals 0.005. Figure x shows the change of pruning time and EDR calculation time under different VC. It can be seen that the most appropriate VC is 2. When CV is bigger than 2, the total execution time shows an increasing trend because of too many EDR calculation caused by the more coarse index. On the other hand, there is also high total time consumption if CV is 0, and it is because more time is spent in generating FVs when the index is too fine. We can see it is important to set a correct CV for acheiving a better query performance.

VII. Related Work

There has been various related works about trajectory storage, indexing and query processing. In this sectoin we review some previous related works about Trajectory Storage and Indexing, Spatial Query Processing.

A. Trajectory Indexing and Query Processing

R-Tree[23] is the most classical index for spatial data, which is a two-dimensional generalization of B-Tree[24]. However, it is not good enough for large-scale trajectory data as the large number of overlappings among the MBR. After that some indices optimized for trajectory such as 3D-RTree[25], TB-Tree[26] and TPR-Tree[27]. They index on the temporal dimension as well as the spatial dimension. However, they are non-adaptive, meaning that they suffered from a performance loss as the diversity of trajectory distribution. SETI[28] proposed a indexing mechanism which forms a two-level index by decoupling the spatial dimension and temporal dimension to reduce the complexity of spatial query. PIST[29] developed a cost model of partitioning the data space for different data distributions, aiming to reduce the number of disk accesses. TrajStore[30] proposed an adaptive algorithm to split trajectories optimally and cluster them physically to achieve a lowest expected cost of query according to the model about the number of pages accessed when executing range queries. Different to the above works which are based on the optimization to the I/O cost, Wang et.al. developed an in-memory column-oriented storage called SharkDB[6] to achieve the high performance in range queries and kNN queries by avoiding the expensive I/O operations. Trajtree[7] was designed to index the computation of EDwP, a similarity metric optimized for trajectories under inconsistent sampling rates. However, these works are all designed to process queries on single-core CPU, so the performance in big trajectory data environment is limited.

There are also some previous works which accelerate various queries on trajectory data using GPU. Zhang et.al[31] proposed a prototype system called U_2STRA to achieve high performance in querying of large-scale trajectory data. It used a four-level hierarchy to split and represent trajectories, and then index them by a grid-file based data structure. Based on this system, he developed TKSimGPU[10], a top-k similarity query algorithm for multicore CPU and manycore GPU, to fill the gap between the importance of this query and the lack of parallel algorithm for it. However, it only support a similarity metric called Hausdorff distance, which is not as popular and robust as some metrics based on global alignment such as EDR[17], DTW[16] and EDwP[7]. Gowanlock et.al[32] developed three GPU-friendly indexing schemes for distance threshold similarity searches for trajectories, in which the segments within an euclidean distance threshold in a point of time are retrieved. Its aims are different from our work, in which all points of time is considered in similarity searches.

B. Parallel and Distributed Spatial Data Analytics

Apart from trajectory, some parallel spatial data analytics frameworks and systems are proposed for spatio-temporal data before. Zhang et.al[11] developed a query processing system to manage big taxi trip data, which record the pickup locations and timestamps of the passengers, by the help of GPU. Lettich et.al[18] designed the PR-quadtree to process stream spatial k-NN queries, in which GPU is used to constructing the index within one second and executing a mass of queries in following seconds to meet the requirement of short response time. Doraiswamy et.al.[8] generalized the kd-tree to STIG, in which points in leaf node are packed to a block to guarantee the fully utilization of GPU when performing interative spatio-temporal queries on it. However, these works are not suitable for managing large-scale trajectory data, because trajectory is not the set but the sequence of spatio-temporal points.

Some works utilize distributed system to solve the efficiency problem in processing spatial queries on single CPU. HadoopGIS [33] adapted and extended Hadoop to meet the challenge of large-scale spatial objects and high computation complexity. It is intergrated with Hive and provides an expressive spatial query language. Lu et.al[34] proposed a solution to the problem of implementing kNN join, an expensive spatial operation for large dataset, in MapReduce to improve the performance of it. Aiming to the performance degradation due to the moving hotspots in distributed spatio-temporal storage system, Pyro[35] adapted HDFS and H-Base by employing a novel DFS block grouping algorithm and multi-scan optimization to reduce the response time of geometry queries.

Based on Spark, Dong et.al developed Simba [5] to provide native support for spatial queries to achieve low query latency and high throughput and excellent scalability. After that, he proposed a distributed framework[36] implemented on Spark which leverage it to answer similarity searches with Hausdorff and Frechet distance as the metric. ???

VIII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] G. Hu, J. Shao, F. Shen, Z. Huang, and H. T. Shen, "Unifying multi-source social media data for personalized travel route planning," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '17. New York, NY, USA: ACM, 2017, pp. 893–896.

[2] K. Zheng, Y. Zheng, N. J. Yuan, S. Shang, and X. Zhou, "Online discovery of gathering patterns over trajectories," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 8, pp. 1974–1988, 2014.

[3] 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.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015, pp. 1383–1394.

[5] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 1071–1085.

[6] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, and S. W. Sadiq, "Sharkdb: An in-memory column-oriented trajectory storage," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, 2014, pp. 1409–1418.

[7] S. Ranu, P. Deepak, A. D. Telang, P. Deshpande, and S. Raghavan, "Indexing and matching trajectories under inconsistent sampling rates," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, Conference Proceedings, pp. 999–1010.

[8] 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.

[9] G. Chen, Y. Ding, and X. Shen, "Sweet KNN: an efficient KNN on GPU through reconciliation between redundancy removal and regularity," in *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, 2017, pp. 621–632.

[10] E. Leal, L. Gruenwald, J. Zhang, and S. You, "Tksimgpu: A parallel top-k trajectory similarity query processing algorithm for gpgpus," in *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, 2015, pp. 461–469.

[11] J. Zhang, S. You, and L. Gruenwald, "High-performance spatial query processing on big taxi trip data using gpgpus," in *2014 IEEE International Congress on Big Data*, 2014, Conference Proceedings, pp. 72–79.

[12] ——, "Parallel online spatial and temporal aggregations on multi-core cpus and many-core gpus," *Inf. Syst.*, vol. 44, pp. 134–154, 2014.

[13] E. Leal, L. Gruenwald, J. Zhang, and S. You, "Towards an efficient top-k trajectory similarity query processing algorithm for big trajectory data on gpgpus," in *2016 IEEE International Congress on Big Data, San Francisco, CA, USA, June 27 - July 2, 2016*, 2016, pp. 206–213.

[14] J. R. Munkres, *Topology*. Prentice Hall, 2000.

[15] M. Vlachos, D. Gunopulos, and G. Kollios, "Discovering similar multidimensional trajectories," in *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, 2002, pp. 673–684.

[16] E. J. Keogh, "Exact indexing of dynamic time warping," in *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, 2002, pp. 406–417.

[17] 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.

[18] F. Lettich, S. Orlando, and C. Silvestri, "Processing streams of spatial k-nn queries and position updates on manycore gpus," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, 2015, pp. 26:1–26:10.

[19] C. Nvidia, "Toolkit documentation," *NVIDIA CUDA Getting Started Guide for Linux*, 2014.

[20] Y. Zheng, L. Zhang, X. Xie, and W. Ma, "Mining interesting locations and travel sequences from GPS trajectories," in *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, 2009, pp. 791–800.

[21] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W. Ma, "Understanding mobility based on GPS data," in *UbiComp 2008: Ubiquitous Computing, 10th International Conference, UbiComp 2008, Seoul, Korea, September 21-24, 2008, Proceedings*, 2008, pp. 312–321.

[22] Y. Zheng, X. Xie, and W. Ma, "Geolife: A collaborative social networking service among user, location and trajectory," *IEEE Data Eng. Bull.*, vol. 33, no. 2, pp. 32–39, 2010.

[23] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, 1984, pp. 47–57.

[24] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," in *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*, 1970, pp. 107–141.

[25] Y. Theodoridis, M. Vazirgiannis, and T. K. Sellis, "Spatio-temporal indexing for large multimedia applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, ICMCS 1996, Hiroshima, Japan, June 17-23, 1996*, 1996, pp. 441–448.

[26] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches to the indexing of moving object trajectories," in *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, 2000, pp. 395–406.

[27] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López, "Indexing the positions of continuously moving objects," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, 2000, pp. 331–342.

[28] V. P. Chakka, A. Everspaugh, and J. M. Patel, "Indexing large trajectory data sets with SETI," in *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.

[29] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander, "PIST: an efficient and practical indexing technique for historical spatio-temporal point data," *GeoInformatica*, vol. 12, no. 2, pp. 143–168, 2008.

[30] P. Cudré-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, 2010, pp. 109–120.

[31] J. Zhang, S. You, and L. Gruenwald, "U2stra: High-performance data management of ubiquitous urban sensing trajectories on gpgpus," in *Proceedings of the 2012 ACM Workshop on City Data Management Workshop*, ser. CDMW '12. New York, NY, USA: ACM, 2012, pp. 5–12.

[32] M. G. Gowanlock and H. Casanova, "Distance threshold similarity searches: Efficient trajectory indexing on the GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2533–2545, 2016.

[33] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce," *PVLDB*, vol. 6, no. 11, pp. 1009–1020, 2013.

[34] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *PVLDB*, vol. 5, no. 10, pp. 1016–1027, 2012.

[35] S. Li, S. Hu, R. K. Ganti, M. Srivatsa, and T. F. Abdelzaher, "Pyro: A spatial-temporal big-data storage system," in *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, 2015, pp. 97–109.

[36] D. Xie, F. Li, and J. M. Phillips, “Distributed trajectory similarity search,” *PVLDB*, vol. 10, no. 11, pp. 1478–1489, 2017.