

GTS: An In-memory GPU-accelerated Trajectory Storage System For Basic Spatial Queries

Bowen Zhang

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

Yanmin Zhu

Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China 200240
Email: yzhu@cs.sjtu.edu.cn

Yanyan Shen

Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China 200240
Email: shen-yy@cs.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-quadtrees 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

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 is 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 private cars in Shanghai, which contains more than 247,494,133 sample points. There are many interesting and valuable information in these data to be exploited, such as route planning[...], traffic prediction[...], travel time prediction [1] etc.. Thousands of queries are produced everyday in these applications. They call 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 it to answer mass of queries in

parallel [2][...]. As a many-core architecture processor, GPU can achieve high throughput by handling thousands of threads at a time.

problem description

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 two basic type of spatial queries: 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, they are both invoked in the process of travel time prediction [1]. In this situation, query delay is significantly important for user's experience. So, it is valuable to develop a trajectory storage system which can make use of GPU to efficiently execute those different kinds of queries in a short response time. However, existing trajectory storage systems are almost optimized for specific kind of query on CPU, causing that applications invoking different kinds of query lose efficiency when running on them. Also, the index designed for CPU cannot be easily migrated to GPU because of the significantly different architecture. 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.

(picture of two kinds of query)

challenge of solving problem

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. This type of queries allows for the usage of triangle inequality because of its metric property. However, global-alignment 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. To solve this problem, it is intuitive to set two database for different kind of query respectively. Unfortunately, this will cause unnecessary space consumption 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 necessary to find a space-saving solution.

(ii) Secondly, there is no proper task division strategy used to improve query performance for some queries oriented to data-level parallelization device such as GPU. We will take EDR, a global-alignment based similarity measurements, as an example. For some simpler similarity measurements such as Hausdorff distance [5], parallelization of similarity query algorithm is easily implemented by dividing whole trajectory into segments first and then assigning each segment to a thread to handle. [5] However, different from this, calculation of EDR is correlated with all points of trajectory, which means a global optimal alignment, disallowing the division of the trajectory in the process of computation.

(iii) Thirdly, because of the architecture limits of GPU, there are many issues should be concerned when designing data layout method and index [2]. For example, memory accessing pattern is a main concern when designing algorithm oriented to GPU, which require us to coalesce the locations of data requests from a warp of threads to a continuous space in global memory of GPU to avoid the high latency when thousands of cores access memory. Sometimes an efficient index working on CPU may disallow this requirement, so it should be specially re-designed if we want to make it work efficiently on GPU.

(I'm finding more challenge...)

existing work about this problem

i. There are some works about accelerating range queries on high-dimensional data with GPU. STIG [2] is an index adapted from traditional kd-tree, which replace point in leaf node with a block consisting a fix number of points to make them able to be checked by GPU in parallel at refine phase of range query. Zhang et.al [6] also propose a platform based on GPU to handle range queries on taxi data, which use adaptive quadtree as its spatial index. However, these two work do not consider the need for similarity query because the focus of them is high-dimensional points rather than trajectories.

ii. TKSImGPU [5] proposes a solution of executing similarity query on GPU. It divides the whole spatial space into thousands of cells, and then segments trajectories according to this division. Finally, parallel computation of Hausdorff distance is achieved on the cell level. However, Hausdorff distance, 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 [7] [8] [9] [10]. Moreover, range query is not supported efficiently by TKSImGPU because of non-adaptive grid.

my idea

In this work, to solve these problems, we design and imple-

ment the GTS (GPU-accelerated Trajectory Storage) system, which can handle both range query and global alignment based top-k similarity query on historical trajectory data. 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. As far as we have known, we are the first one to accelerate top-k similarity query with global alignments by leveraging GPU.

main solutions

i. To support both two types of queries, we propose a novel trajectory storage component. 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-quadtrees to design an index called GT-quadtrees, supporting pruning both on range query and top-k similarity query. In this index the data fetched every time is distributed continuously in memory, so a coalesce access pattern can be achieved conveniently, meaning that this index is GPU-friendly. This index also observes the advantage of quadtree that can be adaptive to different datasets.

ii. To make full use of parallelism power of GPU, we design a special query engine supporting two basic kinds of queries. For range query, we migrate the query algorithm used for quadtree index to GT-quadtrees, 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 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 used in the process of EDR calculation in GPU global memory, which satisfies the requirement of coalesce accessing pattern. We also maintain a buffer table for two types of queries respectively to reduce low-speed data transferring on PCI-E interface, avoiding duplicated data transferring by recording whether trajectory data needed remains in global memory of GPU.

contributions

- We propose a GPU-accelerated in-memory trajectory storage system which support both range query and global alignment based top-k similarity query on a single index.
- We design an quadtree integrated trajectory storage component, not only works for GPU-accelerated range query but also exploiting GPU's parallel power in accelerating top-k similarity query.
- We design a novel solution of accelerating global alignment based similarity query on our storage component by exploiting GPU's parallel power.
- We implement our system on server, evaluate the performance, and prove that our system get 4x speedup for top-k similarity comparing to baseline, and meanwhile, get similar performance on range query comparing to state-of-the-art systems not originally supporting global align-

ment based top-k similarity query, with similar memory consumption.

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 transferred to Euclidean plane.

Definition 2 (trajectory): A trajectory of the object $t = \{p_1, p_2, \dots, p_n\}$ 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 $T = t_1, t_2, \dots, t_{|T|}$, 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 $R_{sim} = \{t_1, t_2, \dots, t_k\}$ 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 adapt to local time shifting may be a good choice. So we use EDR distance as our similarity metric.

Definition 4 (EDR distance [4]): Given two trajectories $t_1 = \{p_1, p_2, \dots, p_{n+1}\}, t_2 = \{p_1, p_2, \dots, p_{n+1}\}$, the EDR

distance between them is calculated by:

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

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

B. GPU computing using CUDA [11]

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 Instruction 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/written 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 throughput. 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 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, outputting 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 opportunities of quadtree with Morton encoding [12] with the consideration of GPGPU 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. Sub-trajectories and quadtree are stored 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 unnecessary 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 supporting 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 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 [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 opportunity 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 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 [12] 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 develop GT-quadtree based on fixed grid index by absorbing some useful features of PR-quadtree. It reaches the goal of achieving load balancing

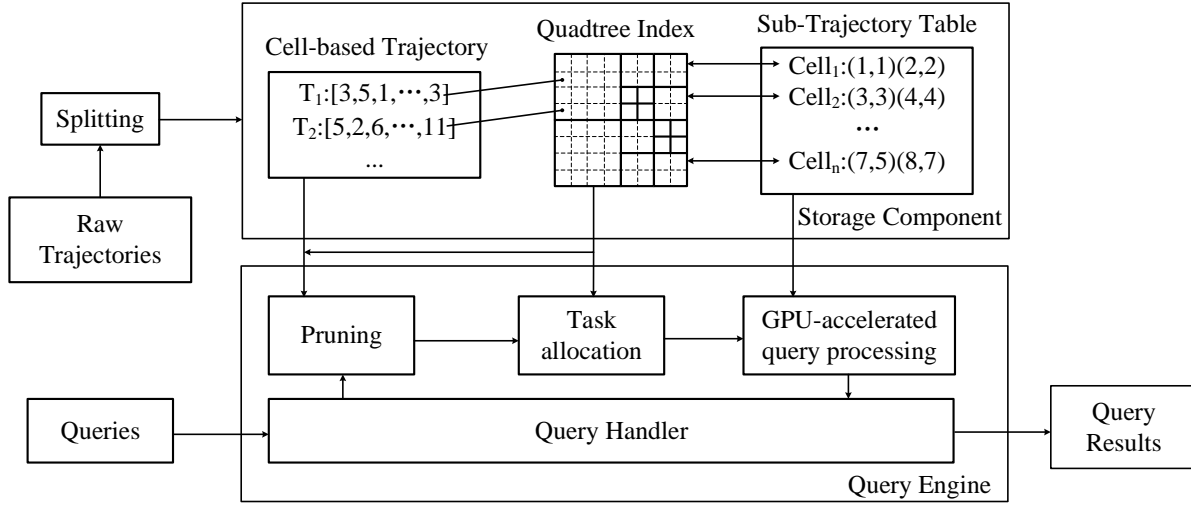


Fig. 1. System Architecture

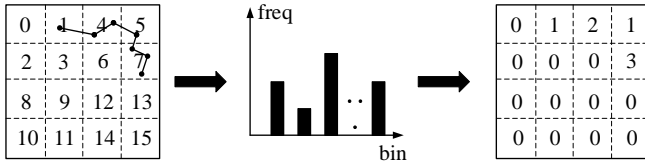


Fig. 2. Example of the trajectory in area (left), the histogram of this trajectory (middle) and the grid that can represent the histogram's value (right)

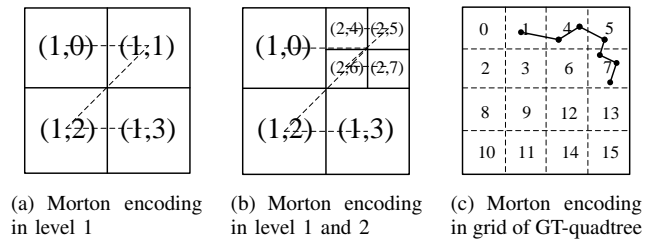


Fig. 3. An example of Morton encoded GT-quadtrees

on GPU, and meanwhile remains the function of serving as the index of similarity query. In GT-quadtrees, each node is corresponding to different number of cells of fixed grid, overlapping 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-quadtrees are 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-quadtrees, 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-quadtrees 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 performing a two-bits right shift on $100(4)$, getting $1(1)$. This feature of GT-quadtrees benefits both range query and similarity query, which will be illustrated in Section V.

We designed sub-trajectories arrays storing the original geographical information of trajectories, making GT-quadtrees meet with the memory access pattern required by GPU. Sub-trajectories array contains the sample points whose lo-

cations are within a given cell in GT-quadtrees'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 $\{p_{1,1}, p_{1,2}, \dots, p_{1,m}\}$ and $\{p_{2,5}, p_{2,6}, \dots, p_{2,n}\}$ from trajectory t_1 and t_2 in cell 15, the content of sub-trajectories array of cell 15 will be $\{p_{1,1}, p_{1,2}, \dots, p_{1,m}, p_{2,5}, p_{2,6}, \dots, p_{2,n}\}$. This arrangement benefits 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 necessary 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-quadtrees. 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 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 equivalent 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 [12], 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

There have been a solution [6] 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.

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 necessary, 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 transferring data, which can hide the latency caused by low-speed PCI to some extent.

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. Refinement procedure on multiple GPUs is similar, so we will not introduce it detailly because of the space limitation.

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. [13][14] Our work can be migrated to other measurements whose

calculation is based on dynamic programming algorithm, such as LCSS [10]. However, considering that GPU efficiency is low if the task load is small, it is suboptimal to directly migrate algorithm proposed by [4] 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.

Algorithm 1 Top-k Similarity Query

Input:

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

Output:

Result list for each query *Result*

```

1: Result  $\leftarrow \text{list}(\text{array}[k])$ 
2: for  $q \in Q$  parallelly do
3:    $LB \leftarrow \text{GenerateLowerBoundParallel}(q, D)$ 
4:    $\text{realEDR} \leftarrow \text{InitialPriorityQueue}()$ 
5:    $\text{candidateSet} \leftarrow D$ 
6:   while  $\text{candidateSet}$  is not empty do
7:      $\text{simiTrajID} \leftarrow \text{ChooseSmall}(LB, \eta)$ 
8:      $\text{realEDR} \leftarrow \text{CalEDRParallel}(\text{simiTrajID})$ 
9:      $\text{candidateSet} \leftarrow \text{FiltParallel}(\max(\text{realEDR}), \text{candidateSet}, LB)$ 
10:  end while
11:   $\text{Result}[q] \leftarrow \text{realEDR.topID}(k)$ 
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 $45 \times 55 \text{ km}^2$ is divided into 247500 cells, if threshold ϵ in EDR is set as $100m$. Besides, although through filtering, massive expensive EDR computations are still necessary to execute. Attention that the computation cost of EDR between two trajectories is $O(mn)$ (m is the length of

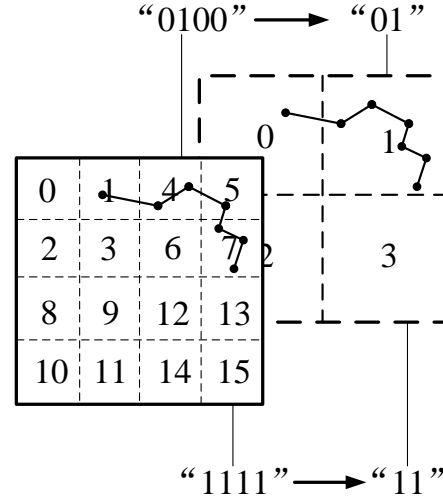


Fig. 4. An example of virtual grid with 4 cells and the transformation between IDs in different grids

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-quadtrees. 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 4, suppose the depth of GT-quadtrees is 3. For a trajectory whose cell-based trajectory is $[1, 4, 4, 5, 7, 7, 7]$, its frequency vector would be $[0, 1, 0, 0, 2, 1, 0, 3, 0, \dots]$ 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, $v_1 = \langle 1, 0, 0, 0 \rangle$, $v_2 = \langle 0, 1, 0, 0 \rangle$, 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 [4], we build a virtual grid, in which the length of cell is larger than that of GT-quadtrees' 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-quadtrees. Figure 4 is an example: a virtual grid with 4 cells is built based on a fixed grid with 16 cells. Chen [4] 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-quadtrees, 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 4, 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.

2) *Parallel EDR Calculation*: 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 actually a dynamic programming, can be divided into several independent steps of comparisons. Figure 5 shows an example of calculating the EDR between two trajectories $traj_1$ 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 minimum 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 5, 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

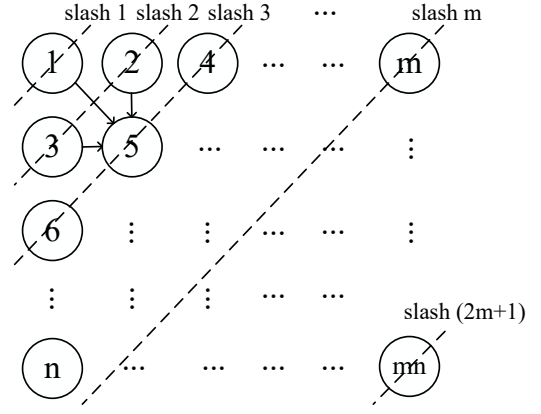


Fig. 5. An example of EDR calculation procedure

GPU full-loading, in order to improve the throughput (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 5, 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 5 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 $(2\max\{m, n\} + 1)$ 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 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 transferring between memory and

Algorithm 2 Parallel EDR Calculation**Input:**

Two sets of trajectories: $T1, T2$
 Threshold of EDR: ϵ
 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**

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 a 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.

TABLE I
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
ϵ	matching threshold of EDR	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

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. 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

2) *Data Preprocessing*: 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 splitting it two meaningful trajectories can be generated. After processing we get at total xxx.....

3) *Parameters*: To systematically test the performance of our system, we conduct our experiments under various parameter settings. Table I 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) *Compared baselines and systems*: For range query, we implement two state-of-the-art systems supporting range query on GPU: STIG [2] and the work of Zhang which using GPU to accelerate sparial query preprocessing on big taxi trip data, noted as WOZ [6]. 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 CPU version GTS as the baseline method.

For similarity query, we only implement the original EDR based top-k similarity query algorithm proposed in [4] on CPU as the baseline because as far as we know we are first to utilize GPU to accelerate EDR based top-k similarity query. All of the queries are executed for 10 times and averages of the time consuming are recorded.

5) *Experiment Overview*: We first test the memory consumption of our system. We measure the average and highest memory usage of different system. The memory consumption is measured after finishing building index for different size dataset, to show that our system use a similar amount of memory comparing to systems not supporting global alignment based top-k similarity query. 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 ($31.11^{\circ}N - 31.36^{\circ}N, 121.39^{\circ}E - 121.58^{\circ}E$). 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 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. Memory Efficiency

to show the memory consumption of our solution is not too larger than state-of-the-art methods. (1 table)

C. Query Latency

Compare with state-of-art methods including STIG and WOZ for range query.

Compare with baseline methods including multi-core CPU implementation of range query and top-k similarity query.

TKSimGPU use a simpler similarity defination method, so we don't consider it in our experiment.

1) Range Query:

(figure 6): different N_Q (workload)

2) Top-k similarity Query:

(figure 11): different N_Q (workload)

D. Speedup Ratio

Show how query latency changes as the number of GPU goes from 1 to 2 (use the GPU of DDST). And the number of cores used can not be controlled in CUDA.

(two figures) Compare with single-core CPU, the speedup ratio

different GPU num

E. Scalability

1) Range Query:

(figure 7)different dataset size

2) Top-k similarity Query:

(figure 12): different dataset size

F. Parameters Tuning

1) Range Query:

(figure 3): different L_{cell}

(figure 4): different M_{cell}

(figure 5): different S_{RQ} (selectivity)

2) Top-k similarity Query:

(figure 8): different L_{cell}

(figure 9): different ϵ

(figure 10): different k

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] 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. [Online]. Available: <https://doi.org/10.1109/BigData.2015.7363787>

- [6] 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.
- [7] S. Ranu, D. P. 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*, April 2015, pp. 999–1010.
- [8] L. Chen and R. T. Ng, "On the marriage of lp-norms and edit distance," in *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, 2004, pp. 792–803. [Online]. Available: <http://www.vldb.org/conf/2004/RS21P2.PDF>
- [9] B. Yi, H. V. Jagadish, and C. Faloutsos, "Efficient retrieval of similar time sequences under time warping," in *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, 1998, pp. 201–208. [Online]. Available: <https://doi.org/10.1109/ICDE.1998.655778>
- [10] 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. [Online]. Available: <https://doi.org/10.1109/ICDE.2002.994784>
- [11] C. Nvidia, "Toolkit documentation," *NVIDIA CUDA Getting Started Guide for Linux*, 2014.
- [12] 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.
- [13] 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.