A GPU-accelerated Framework for Processing Trajectory Queries

Bowen Zhang, Yanyan Shen, Yanmin Zhu and Jiadi Yu
Department of Computer Science and Engineering, Shanghai Jiao Tong University
Email: {zbw0046, shenyy, yzhu, jiadiyu}@sjtu.edu.cn

***Abstract*—The increasing amount of trajectory data facilitates a wide spectrum of practical applications. In many such applications, large numbers of trajectory range and similarity queries are issued continuously, which calls for high-throughput trajectory query processing. Traditional in-memory databases lack considerations of the unique features of trajectories, thus suffering from inferior performance. Existing trajectory query processing systems are typically designed for only one type of trajectory queries, i.e., either range or similarity query, but not for both. Inspired by the massive parallelism on GPUs, in this paper, we develop a GPU-accelerated framework, named GAT, to support both types of trajectory queries (i.e., both range and similarity queries) with high throughput. For similarity queries, we adopt the Edit Distance on Real sequence (EDR) as the similarity measure which is accurate and robust to noise in real-world trajectories. GAT employs a GPU-friendly index called GTIDX to effectively filter invalid trajectories for both range and similarity queries, and exploits the GPU to perform parallel verifications. To accelerate the verification process on the GPU, we apply the Morton-based encoding method to reorganize trajectory points and facilitate coalesced data accesses for individual point data in global memory, which reduces the global memory bandwidth requirement significantly. We also propose a technique of grouping size-varying cells into balanced blocks with similar numbers of trajectory points, to achieve load balancing among the Streaming Multiprocessors (SMs) of the GPU. We conduct extensive experiments to evaluate the performance of GAT using two real-life trajectory datasets. The results show that GAT is scalable and achieves high throughput with acceptable indexing cost.**

I. Introduction

Recent years have witnessed a surge of trajectories being continuously generated from every corner around the world. For instance, DiDi Chuxing, China's largest online ride-sharing platform, processes over three million trip requests every day, suggesting that thousands of trajectories are generated in every second [1]. A *trajectory* is typically represented by a sequence of successive points of moving objects, where each point consists of geospatial coordinates and the corresponding timestamp. Large trajectory data facilitate a wide spectrum of real-world applications, such as route planing [2], trajectory pattern mining [3] and travel time prediction [4]. In these applications, two types of trajectory queries, i.e., *range query* and *similarity query*, serve as primitive, yet essential operations.

A range query aims to identify trajectories overlapping a given bounding area, while a similarity query determines the k trajectories that are most similar to the query trajectory (the similarity is typically measured by a certain distance function). Both types of trajectory queries are expensive to compute, in face of hundreds of millions of trajectories. Furthermore, in many real-world applications, large numbers of trajectory queries may be continuously generated and should be efficiently processed in order to fulfill application requirements. Take the taxi fleet management in Shanghai for example. Detour detections pose top-k similarity queries on historical trajectories to determine if the ride trajectory of a given taxi is an outlier, comparing against the historical trajectories sharing similar origins and destinations. And lost found services raise range queries in order to locate all taxis that traversed in a given region. There are around 50,000 taxis in Shanghai, the largest metropolitan in China, and hundreds of range and top-k similarity queries may be generated in every minute, which should be processed efficiently in support of the management of such a large taxi fleet. *Therefore, processing trajectory queries in high throughput has become significantly important.*

While traditional in-memory databases (e.g., HStore [5]) or data processing frameworks (e.g., SparkSQL [6]) provide generic solutions that can process the two types of trajectory queries, their performance is typically suboptimal, due to the lack of considerations on the unique features of trajectories. Recently, considerable research efforts have been devoted to developing specialized trajectory query processing systems (e.g., Simba [7], SharkDB [8]). However, these existing solutions are typically designed for answering only a certain type of queries, either range or similarity queries, but not both. To process both types of queries together, a possible extension to these solutions is to build separate indices with different parameter settings for each type of queries, but resulting in problems such as extra memory cost.

To scale up the performance, a natural solution is to exploit GPUs to accelerate trajectory query processing. GPUs have recently been widely applied in some practical applications, thanks to their massive parallelism. To the best of our knowledge, however, only one recent work [9] attempted at leveraging GPUs for processing trajectory similarity queries. They employ the Hausdorff distance function [10] for measuring the similarity between two trajectories. This distance function can easily be decomposed into independent computations, hence facilitating the parallel processing on the GPU. However, the major disadvantages of this work are three-fold. First, the underlying Hausdorff distance function is sensitive to location noise and local time shifting (which is common in real-world trajectories), which may return some trajectories that are not similar to the query trajectory [11]. Second, its parallel query processing on the GPU is highly coupled with the Hausdorff distance function, making it inextensible to more realistic distance functions such as EDR [12]. Finally, it is designed for only one type of queries, i.e., similarity queries, and cannot be easily extended for efficient processing of range queries.

In this paper, we develop a GPU-accelerated framework, named GAT, for processing both the two types of trajectory queries (i.e., both range and similarity queries). To return high-quality similar trajectories, we adopt the EDR [12] as the distance function, which is an accurate and robust similarity measure for searching for similar trajectories [11]. Generally, GAT follows the filtering-and-verification framework where the CPU is responsible for generating candidate trajectories via effective pruning rules and the GPU performs parallel candidate verifications for final resulting answers. The filtering basis of GAT is a GPU-friendly index called GTIDX that fuses the merits of the cell-level trajectory representations and the quadtree-like index, and reduces the number of candidates for both types of trajectory queries effectively. The candidate trajectories are then be transferred to the GPU for verifications. We find that there exists overlap among the candidates of different queries. This inspires us to propose a Memory Allocated Table (MAT) for checking the existence of data in global memory of the GPU, thus avoiding redundant data transfers over PCIe bus. During verification, for range queries, we leverage the GPU to determine whether a candidate trajectory overlaps a given bounding area in parallel; For top-k similarity queries, we iteratively supply the GPU with a set of candidate trajectories based on the lower bound of EDR distance, parallelize each EDR distance computation, and progressively refine a priority queue that maintains the possible top-k answers according to the computed EDR values.

To exploit GPUs for efficient candidate verification, we have to address two practical issues. The first issue is the *limited global memory bandwidth* between main memory and global memory of the GPU. Threads running on different GPU cores access trajectory data from the shared global memory concurrently. Numerous individual data requests from thousands of cores may easily compromise the verification performance due to the limited bandwidth of global memory. The second issue is *load balancing among the Streaming Multiprocessors (SMs) of the GPU*. The GPU follows the Single Instruction Multiple Data parallelism model and it is critical to assign similar amount of computational workload among the SMs on the GPU for high processing performance. Any SMs with heavier workloads may become stragglers, hindering the overall performance.

In GAT, we provide a data placement strategy to facilitate *coalesced global memory data access* among GPU cores. We apply the Morton-based encoding method [13] to reorder a set of cells and reorganize trajectory points according to these cells. The Morton's codes ensure that trajectory points to be verified by the GPU cores in the same SM are stored continuously in global memory, and the individual accesses to these points can be coalesced into a single access request to significantly reduce the global memory bandwidth requirement. To achieve load balancing, we deliberately *group size-varying cells into balanced blocks*, where different balanced blocks involve similar numbers of trajectory points. By assigning the balanced blocks with similar numbers of trajectory points to the SMs, we avoid overloading some SMs to a great extent.

The main contributions of this work are the following.

• We develop a GPU-accelerated framework GAT to support the two types of queries, including both range and top-k similarity queries over trajectories. GAT follows the filtering-and-verification framework, where we identify candidates on the CPU and leverage the GPU to perform parallel verifications to increase the overall throughput of query processing.

• We design a GPU-friendly index GTIDX that organizes trajectory points into geospatial cells, which is effective in pruning invalid candidates for both types of trajectory queries and generating similar-sized candidates for achieving load balance among the SMs on the GPU.

• It is the first work, to the best of our knowledge, that adopts the EDR distance function for processing similarity queries using the GPU. The EDR distance function is accurate for measuring the similarity between trajectories, and robust to location noises. We decompose the EDR calculation into independent components to exploit the GPU.

• We propose two techniques of accelerating the verification process on the GPU. We employ the Morton-based encoding to facilitate coalesced individual accesses from cores to reduce the global memory bandwidth requirement. And we propose a technique of grouping size-varying cells into balanced blocks to achieve load balancing among the GPU SMs.

• We conduct extensive experiments to evaluate the performance of GAT using two real-life trajectory datasets. The results show that our framework is efficient, scalable and with high throughput. It achieves up to 38x and 75x speedups for trajectory range and top-k similarity queries respectively, compared with CPU-based single-threading solutions, and outperforms the adaptations of GPU solutions for range queries.

The remainder of this paper is organized as follows. Section II gives preliminaries. Section III provides an overview of GAT. Section IV presents the index structure. Section V provides GPU-accelerated query processing approaches. We show experimental results in Section VI, review related works in Section VII and conclude this paper in Section VIII.

II. Preliminaries

A. Trajectory Range and Similarity Queries

Definition 1 (Trajectory T): A trajectory T is a sequence of points that captures the trace of a moving object over time, i.e., $T = [(p_1, t_1), \cdots, (p_n, t_n)]$, where $p_i = (x_i, y_i)$ records the geospatial coordinates and t_i is a timestamp, for $i \in [1, n]$. Let $|T|$ denote the number of points in T and $\mathcal{T}$ be the set of all trajectories.

Definition 2 (Trajectory Range Query $\mathcal{Q}_r$): Given a query region R and a trajectory dataset $\mathcal{T}$, a range query $\mathcal{Q}_r(R)$ returns a set of trajectories overlapped with R, i.e.,

$$\mathcal{Q}_r(R) = \{T \in \mathcal{T} | \exists p_i \in T,\ s.t.\ p_i \in R\} \tag{1}$$

For simplicity, we consider query regions as two-dimensional rectangles in this paper, but our approach can be adapted to handle regions in arbitrary shapes.

We use *Edit Distance on Real sequence* (EDR) to measure the similarity between two trajectories, which is robust and accurate than other distance functions [12].

Definition 3 (EDR δ): Given two trajectories T and T', the EDR between T and T' is the minimum number of insert, delete or replace operations needed to transform T to T':

$$\delta(T,T') = \begin{cases} |T| & \text{if } |T'| = 0 \\ |T'| & \text{if } |T| = 0 \\ \min\{\delta(\bar{T},\bar{T}') + subcost, \\ \delta(\bar{T},T')+1, \delta(T,\bar{T}')+1\} & \text{otherwise} \end{cases} \tag{2}$$

where $\bar{T} = [(p_2,t_2),\cdots,(p_n,t_n)]$, (resp. $\bar{T}'$), and $subcost = 0$ if $|x_1 - x'_1| \leq \epsilon \wedge |y_1 - y'_1| \leq \epsilon$, or 1 otherwise. ϵ is a matching threshold.

Definition 4 (Top-k Trajectory Similarity Query $\mathcal{Q}_k$): Given $\mathcal{T}$, a query trajectory T_q and a positive integer k, the top-k trajectory similarity query $\mathcal{Q}_k$ return the k most similar trajectories in $\mathcal{T}$, denoted by $\mathcal{Q}_k(T_q) \subseteq \mathcal{T}$, satisfying: $\forall T' \in \mathcal{Q}_k(T_q)$, $\forall T'' \in \mathcal{T} - \mathcal{Q}_k(T_q)$, $\delta(T_q,T') \leq \delta(T_q,T'')$.

B. GPUs

A GPU typically consists of tens of SMs where each SM includes a number of cores running threads concurrently. All threads execute the same set of instructions over different data, following the Single Instruction Multiple Data (SIMD) parallelism model. Each SM has its local memory and multiple cores, and all SMs share global memory. Data used by threads are usually transferred from main memory to global memory first, via PCIe bus (the typical bandwidth is lower than 10GB/s). We adopt CUDA [14] proposed by NVIDIA as our programming model over GPUs. A GPU program is also referred to as a *kernel*. When executing a kernel program, a grid of CUDA threads are launched immediately. All CUDA threads are divided into blocks and further into warps explicitly. During GPU computing, each CUDA block is assigned to on SM which maintains a warp scheduler to determine which warp to be executed on the cores in the SM in the next clock cycle.

III. Overview of GAT

The overview of GAT is depicted in Figure 1, which consists of two main components, i.e., Cell-based Index Construction and Trajectory Query Processing (including Range Query Processing and Similarity Query Processing).

A. Cell-based Index Construction

There are two main steps in this component:

• **Reorganizing trajectory points**: As discussed in Section I, our index structure depends on cell-level trajectory representation. This is archived by constructing a square called *whole region* covering all the trajectory points. We divide the whole region into 4^n disjoint *cell*s of the same size along the two geospatial coordinates, where n can be pre-defined to

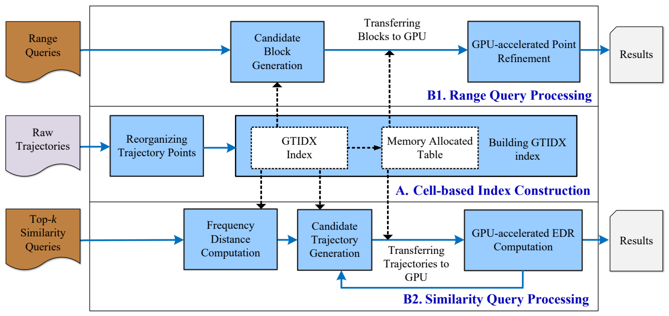

Range Queries
Candidate Block Generation
Transferring Blocks to GPU
GPU-accelerated Point Refinement
Results
B1. Range Query Processing
Raw Trajectories
Reorganizing Trajectory Points
GTIDX Index
Memory Allocated Table
Building GTIDX index
A. Cell-based Index Construction
Top-k Similarity Queries
Frequency Distance Computation
Candidate Trajectory Generation
Transferring Trajectories to GPU
GPU-accelerated EDR Computation
Results
B2. Similarity Query Processing

Fig. 1. The overview of GAT

control the total number of cells. Given the raw trajectories, we map each trajectory point to its covering cell. All the points in the same cell are sorted by their belonging trajectory IDs and timestamps in ascending order. Let $\mathcal{C}$ denote all the cells and $P(C)$ denote a sequence of points residing in cell $C \in \mathcal{C}$.

• **Building GTIDX index**: Our index structure contains three parts: the cell-based trajectory table, the trajectory index and the quadtree-like index. The cell-based trajectory table stores all the points organized by cells, acting as the storage of raw trajectories. The trajectory index is used to reconstruct trajectories from the cell-based trajectory table, which facilitates efficient processing of top-k similarity queries. The quadtree-like index organizes cells into *block*s. Each leaf node in this index corresponds to a block. While the number of cells in each block may differ, the number of the involving trajectory points per block is similar. Organizing cells into blocks helps achieve load balanceing for efficient query processing.

B. Trajectory Query Processing

Without loss of generality, we illustrate the major steps of trajectory query processing by considering a single query. The processing of different queries is independent and concurrent. We first introduce how to perform trajectory range queries, which involves two steps.

• **Candidate block generation**: The input to this step is a query region R and the output is a set of blocks containing several cells with candidate trajectory points, denoted by $\mathcal{Q}_r^c$. First, we traverse the quadtree-like index to locate leaf nodes where the corresponding blocks overlap R. Then, these blocks are identified as candidate blocks and put into $\mathcal{Q}_r^c$. If the points within each of the candidate blocks are not yet in global memory, we transfer the points to global memory and the corresponding items in the MAT are updated.

• **GPU-accelerated point verification**: In this step, we verify whether each point in the candidate blocks $\mathcal{Q}_r^c$ is within the query region using GPU. If a point resides in R, we retrieve the corresponding trajectory based on trajectory ID and add it into the final result set $\mathcal{Q}_r(R)$. For load balance purpose, we assign a block of trajectory points to one SM for verification.

To handle trajectory top-k similarity queries, we maintain a k-sized priority queue Q_k^r for k trajectories with the smallest computed EDR distances, and perform the following three steps, where Step 2 and 3 are executed iteratively.

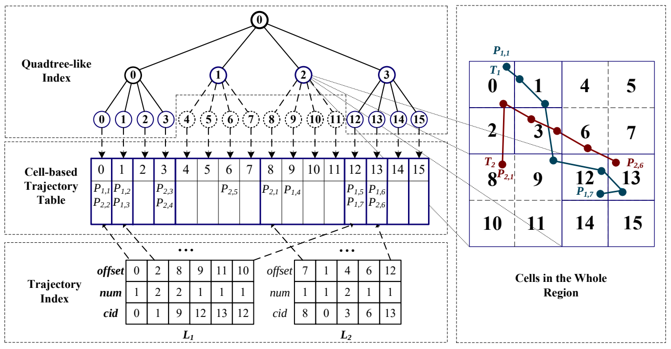


Fig. 2. The structure of GTIDX

• **Frequency distance (FD) computation**: The input to this step is a query trajectory T_q and a positive integer k. In this step, we compute a *lower bound* of $\delta(T_q, T)$ [12] for each $T \in \mathcal{T}$, which will be used for reducing unnecessary EDR computations.

• **Candidate trajectory generation**: In each iteration of this step, we select a set of trajectories with the smallest EDR lower bounds, because they are more likely to be in the top-k results. We then reconstruct these candidate trajectories, denoted by $\mathcal{Q}_k^c$, from the points stored in cells via our index.

• **GPU-accelerated EDR computation**: In each iteration, the input to this step is a query trajectory T_q and candidate trajectories $\mathcal{Q}_k^c$, to compute EDR distances for all the candidates. We distribute EDR computations among CUDA blocks, and each EDR calculation is decomposed into independent components to be performed by CUDA threads concurrently.

At the end of each iteration, we keep the IDs of k trajectories with the smallest EDR distances computed so far using priority queue Q_k^r. The iteration terminates iff all the EDR lower bounds for the unchecked trajectories are larger than the k distances of trajectories belonging to Q_k^r. Finally, we put the k trajectories based on the IDs in Q_k^r into $\mathcal{Q}_k$.

IV. GPU-FRIENDLY CELL-BASED INDEX

A. *The Structure of GTIDX*

Our index GTIDX is developed based on a set of cells, covering all the trajectories. Figure 2 shows an example of 16 cells in the whole region including two trajectories marked with red and blue lines. Each cell C has an identifier cid, which is determined by the Morton-based encoding method [13].

We develop our index GTIDX following a three-part structure: cell-based trajectory table $\mathcal{S}$, trajectory index $\mathcal{I}_T$ and quadtree-like index $\mathcal{I}_Q$. Figure 2 shows the index structure.

1) Cell-based trajectory table $\mathcal{S}$: This table is used to store all trajectories in cell-level representation. We adopt such representation to facilitate pruning strategies for both types of queries. To get the cell-level representation for any trajectory in $\mathcal{T}$, we map each trajectory point to its covering cell. We then define a cell-based trajectory table $\mathcal{S}$ as a sequence of cells in ascending order of cell identifier. Within each cell $C \in \mathcal{C}$, we keep all the trajectory points covered by it, denoted by $\phi_p(C)$, in ascending order of (trajectory ID, timestamp). In this manner, $\mathcal{S}$ stores points from the same trajectory in each cell continuously.

2) Trajectory index $\mathcal{I}_T$: We associate the table S with a trajectory index $\mathcal{I}_T$ to reconstruct any trajectory efficiently. The index maps a trajectory ID tid to a vector L_{tid}. Each element in L_{tid} is a triplet of $(cid, offset, length)$, representing a subsequence of the raw trajectory T_{tid}. Specifically, $offset$ and $length$ specify a continuous part of $\mathcal{S}$ starting from $offset$ to $offset+length-1$, in which all the points in the subsequence are within C_{cid}. By traversing all the elements in L_{tid}, we are able to obtain all the points in trajectory T_{tid}. Note that the size of L_{tid} is typically smaller than the number of trajectory points, especially when the points from a trajectory are aggregated into a few cells.

Figure 2 shows an example of $\mathcal{S}$ and $\mathcal{I}_T$ for two trajectories T_1 and T_2. We take T_1 as an example. The seven points of T_1 are split into five cells, and the points within each cell are stored in the corresponding item of $\mathcal{S}$. Six subsequences can be extracted from T_1 according to $\mathcal{C}$, creating six elements in L_1. For instance, the second element $(1, 2, 2)$ indicates that this subsequence have two points in C_1 that are stored in $\mathcal{S}[2]$.

3) Quadtree-like index $\mathcal{I}_Q$: While the cell-based trajectory table provides a way to organize trajectory points, it is easy to see that the number of points per cell may differ greatly. Such differences result in unbalanced workloads among GPU SMs during query processing (see details in Section V). To alleviate the problem, we propose to group cells into *balanced block*s using a quadtree-like index $\mathcal{I}_Q$, which can be viewed as a subtree of the complete quadtree QT, as described below.

QT organizes all the 4^n cells using $n+1$ levels and each node in it corresponds to a *block* of cells, e.g., the root node corresponds to the whole set of cells, and each leaf node is a block with one cell. We define the number of points covered by the cells in a block as the *size* of this block. A node in QT is denoted by $N = (l, nid, isLeaf)$, where l is the node level identifier from 0 to n, nid is the node ID in this level and $isLeaf$ is a binary variable indicating whether N is a leaf or not. Note that (l, nid) uniquely identifies a node in QT. Interestingly, by employing Morton codes as cell identifiers, we can directly compute the set $\phi_c(N)$ of cells covered by any node N in QT:

$$\phi_c(N) = \{C \in \mathcal{C} \mid 4^{n-l} \times nid \le C.cid < 4^{n-l} \times (nid+1)\} \quad (3)$$

The above equation is guaranteed by the well-known relationship between Morton codes and quadtree structural properties [15]. More importantly, the cells in $\phi_c(N)$ reside continuously in the trajectory table $\mathcal{S}$.

We construct $\mathcal{I}_Q$ by pruning QT in a bottom-up way. That is, we recursively prune all the child nodes of N iff the total number of trajectory points covered by the child nodes is lower than a threshold θ_p called the *block size threshold*. The resulting tree is recognized as $\mathcal{I}_Q$ with a set of new leaf nodes, denoted by $\phi_l(\mathcal{I}_Q)$, which cannot be pruned further. For instance, in Figure 2, we set $\theta_p = 3$, then the new leaf nodes of $\mathcal{I}_Q$ are $\{(1, 1, true), (1, 2, true)\}$ as the total numbers of points in the child nodes for each of them are below three. One can verify that the new leaf nodes in $\mathcal{I}_Q$ provide a grouping over cells and they are the only nodes in $\mathcal{I}_Q$ covering less than θ_p trajectory points:

$$\bigcup_{N \in \phi_l(\mathcal{I}_Q)} \phi_c(N) = \mathcal{C} \wedge \phi_c(N) \cap \phi_c(N') = \emptyset, \forall N, N' \in \phi_l(\mathcal{I}_Q) \quad (4)$$

$$|\phi_p(N)| < \theta_p, \forall N \in \phi_l(\mathcal{I}_Q) \wedge |\phi_p(N)| \geq \theta_p, \forall N \in \mathcal{I}_Q - \phi_l(\mathcal{I}_Q) \quad (5)$$

Note that the original quadtree QT satisfies two properties: (1) the nids of lead nodes in QT have a 1-to-1 mapping to cell identifiers cids; (2) if node N' is the parent of N, we have $N'.level = N.level - 1 \wedge N'.nid = \lfloor N.nid/4 \rfloor$. Based on these properties, for any cell $C \in \mathcal{C}$, we can compute the leaf node in $\mathcal{I}_Q$ covering cell C based on the following theorem.

Theorem 1: Given $\mathcal{I}_Q$ over $\mathcal{C}$ and a cell $C \in \mathcal{C}$, a leaf node $N \in \phi_l(\mathcal{I}_Q)$ covers C iff $\lfloor C.cid/4^{n-N.l} \rfloor = N.nid$.

Benefits of GTIDX. There are three main benefits of GTIDX: (1) the cell-based trajectory table $\mathcal{S}$ facilitates the fast pruning of invalid trajectories for range queries and the computation of EDR lower bounds for similarity queries. And the light-weight index $\mathcal{I}_T$ enables fast reconstruction of trajectories; (2) in each cell, the points from the same trajectory are stored continuously in S, which enables coalesced data accesses in global memory; (3) quadtree-like index $\mathcal{I}_Q$ groups size-varying cells into balanced blocks. This grouping balances the verification workload among GPU SMs for range queries.

Space complexity. We now discuss the space requirement of GTIDX. First, the trajectory table $\mathcal{S}$ contains all the trajectory points and hence the space cost of S is $O(\sum_{T \in \mathcal{T}} |T|)$. Second, trajectory index $\mathcal{I}_T$ maintains a vector of $(cid, offset, length)$ triplets for each trajectory in $\mathcal{T}$. Thus, the space cost of $\mathcal{I}_T$ is $O(\sum_{T \in \mathcal{T}} |T|)$. Third, the tree index $\mathcal{I}_Q$ contains at most 2×4^n nodes with the fields of $(l, nid, isLeaf)$. Hence, the space cost of $\mathcal{I}_Q$ is $O(4^n)$. In practice, $4^n \ll \sum_{T \in \mathcal{T}} |T|$ and the total number of triplets in $\mathcal{I}_T$ is much smaller than that of trajectory points. These two factors make the actual space requirement of GTIDX close to the size of trajectories. Note that we do not need to keep the original trajectories as $\mathcal{S}$ has already contained all the data. Hence, GTIDX is space-efficient.

B. GTIDX Construction

Algorithm 1 provides the details of constructing GTIDX. To compute $\mathcal{S}$ and $\mathcal{I}_T$, we traverse all the trajectories in $\mathcal{T}$ (line 1). Given a trajectory T, we first split T into a set of subsequence $\mathcal{M}$, ensuring points in each subsequence belong to the same cell (line 2). For each subsequence M in $\mathcal{M}$, we append a triplet to $\mathcal{I}_T[T.tid]$ recording the information that $|M|$ points in M from trajectory T are covered by cell cid (line 4). We also update $\mathcal{S}$ accordingly (line 5).

Next, we construct a complete quadtree QT (line 8) and perform a bottom-up pruning to obtain $\mathcal{I}_Q$ (line 9-23). We start from the nodes in the deepest level of QT (line 9). If the cells covered by a node N contain less than θ_p points, we prune its child nodes in QT (line 15) and mark N as a leaf node (line 16). The process is repeated from level n to level 0 of QT and we perform early termination if no pruning is conducted at a certain level (line 19-21).

Algorithm 1: Index Construction

```
   Input: raw trajectories $\mathcal{T}$, $n$, $\theta_p$
   Output: $\mathcal{S}$, $\mathcal{I}_T$, $\mathcal{I}_Q$
1  for $tid \in [1, |\mathcal{T}|]$ do
2      $\mathcal{M} \leftarrow$splitTraj($T, n$);
3      for $M \in \mathcal{M}$ do
4          $\mathcal{I}_T[tid]$.insert($cid, \mathcal{S}[cid].size(), |M|$);
5          $\mathcal{S}[cid] \leftarrow \mathcal{S}[cid] \cup M$;
6      end
7  end
8  $QT \leftarrow$buildFullQuadTree($n$);
9  for $l = n$ to $0$ do
10     $terminate \leftarrow True$;
11     for $nid = 0$ to $4^l - 1$ do
12         $N \leftarrow$getNode($QT, l, nid$);
13         if $|\phi_p(N)| < \phi_p$ then
14             $terminate \leftarrow False$;
15             pruneChildren($QT, N$);
16             $N.isLeaf \leftarrow True$;
17         end
18     end
19     if $terminate = True$ then
20         break;
21     end
22 end
23 $\mathcal{I}_Q \leftarrow QT$;
```

V. GPU-accelerated Query Processing

A. Motivation and Overview

GAT follows the filtering-and-verification framework, where the filtering step prunes invalid trajectories and the verification step verifies all the candidates to get the final results. While various spatial indices [16] [17] have been proposed to reduce the number of candidate trajectories, it is worth mentioning that the filtering step is usually efficient and represents a small fraction of the overall query processing time. In contrast, the verification step is typically very expensive to compute. For range queries, the number of candidate trajectories is determined by the size and the location of the query region R, which can be huge especially in the presence of batch queries. For top-k similarity queries, the quadratic time complexity of EDR computation may become the bottleneck of query processing. We hence focus on accelerating candidate verification. An important observation is that *the verification step involves executing the same procedure over different data*. For instance, we need to check whether each of the candidate trajectory points is within a query region or not, or compute EDR distances for a large number of trajectory pairs. This possibility of data-level parallelism inspires us to leverage GPU techniques to speed up the verification step.

A straightforward way to perform verifications for batch queries on GPUs is to treat each query independently using one CUDA thread. However, the verification workloads among different queries may vary a lot and the overlap of candidate trajectory data accessed by many queries may cause redundant

Algorithm 2: Range Query

```
   **Input:** a set of query regions $\{R\}$, $\mathcal{S}$, $\mathcal{I}_T$, $\mathcal{I}_Q$;
   **Output:** a result set of trajectories $\{\mathcal{Q}_r(R)\}$;
1  for $R \in \{R\}$ parallelly do
2      // (1) filtering by CPU
3      $Q_c \leftarrow$ createQueue();
4      $Q_c$.push($\mathcal{I}_Q.root$);
5      while $Q_c \neq \phi$ do
6          $N \leftarrow Q_c$.pop();
7          $isOverlap \leftarrow$ checkOverlap($R, N$);
8          if $isOverlap = True$ and $N.isLeaf = True$ then
9              $\mathcal{Q}_r^c$.insert(extractBlock($N$));
10         end
11         if $isOverlap = True$ and $N.isLeaf = False$ then
12             $\{N_{child}\} \leftarrow$ findChildNodes($N$);
13             $Q_c$.push($\{N_{child}\}$);
14         end
15     end
16     copyBlockToGPU($\mathcal{Q}_r^c, \mathcal{S}$);
17     // (2) verification by GPU
18     $\mathcal{B}_r \leftarrow$ initBitMap($|\mathcal{T}|$);
19     for CUDA block $bID \in [0, size(\mathcal{Q}_r^c) - 1]$ parallelly do
20         $\mathcal{P} \leftarrow$ extractPoints();
21         for $i \in [0, size(\mathcal{P})/N_{th} - 1]$ do
22             for CUDA thread $thID \in [0, N_{th} - 1]$ parallelly
                do
23                 $p \leftarrow \mathcal{P}[i * N_{th} + thID]$;
24                 $\mathcal{B}_r[p.tID] \leftarrow$ pointVerify($p$, $R$);
25             end
26         end
27     end
28     $\mathcal{Q}_r(R) \leftarrow$ recoverTraj($\mathcal{B}_r, \mathcal{S}, \mathcal{I}_T$);
29 end
```

data transfer between main memory and global memory. Both factors reduce query processing throughput significantly. A fine-grained task assignment is thus required to address the problem. In what follows, we present our GPU-accelerated method for trajectory query processing.

B. Trajectory Range Query

Algorithm 2 illustrates the pseudo code of trajectory range query processing in GAT. We perform range queries in two steps: (1) filtering invalid trajectories by CPU and (2) verifying candidates by GPU. For Step 1, we traverse the tree index $\mathcal{I}_Q$ to identify the blocks spatially overlapping the query region. We define that a block *overlaps* a region if there exists at least one cell in this block overlapping the region. It is better to use the CPU to perform the filtering due to the complex branch instructions involved in this step.

1) Filtering by CPU: The input to this step is a set of query regions $\{R\}$ of the range queries $\{\mathcal{Q}_r\}$ and the output is a set $\mathcal{Q}_r^c$ of candidate blocks (i.e., sets of cells) overlapping the query region R for each $\mathcal{Q}_r$. Each block in $\mathcal{Q}_r^c$ corresponds to a leaf node in $\mathcal{I}_Q$ and spatially overlaps with the query region R. The points covered by a candidate block in $\mathcal{Q}_r^c$ will be processed by a GPU SM for verification. We consider leaf nodes since they are balanced in size and each of them contains less than θ_p points according to Formula 5. For illustration purpose, we first describe how to perform filtering for one

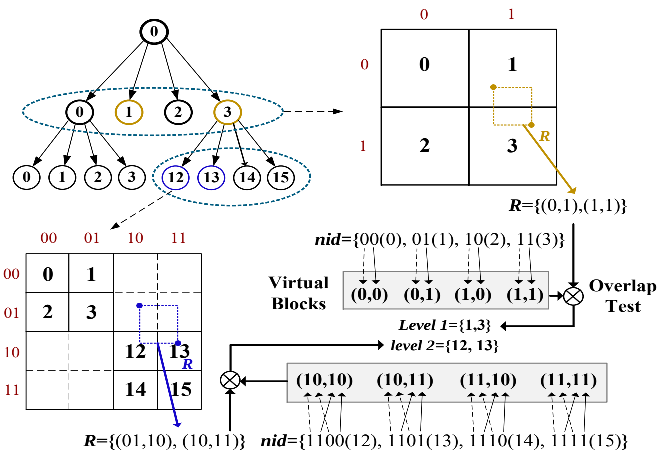


Fig. 3. An example of filtering in processing range queries

range query $\mathcal{Q}_r$. We denote by (r_0, r_1) the top-left and bottom-right points of the query region R.

A naïve way to computing $\mathcal{Q}_r^c$ for $\mathcal{Q}_r$ is to check all the leaf nodes in $\mathcal{I}_Q$. For a leaf node $N \in \phi_l(\mathcal{I}_Q)$, if the corresponding block overlaps the query region R, we add the block to $\mathcal{Q}_r^c$. However, we observe that the query region is typically small and most leaf nodes in $\mathcal{I}_Q$ are not candidates. To speed up the process, we perform a breath-first search over $\mathcal{I}_Q$ (line 4-15) and apply the following pruning rule (line 7) during traversing, to avoid examining descendants of a node that is not a candidate.

Theorem 2: Given any node N in $\mathcal{I}_Q$, if the corresponding block does not overlap the query region R, all the blocks corresponding to the descendants of N are not candidates.

The only missing part of our filtering step is how to quickly determine whether a block overlaps R. In $\mathcal{I}_Q$, by applying the Morton-based encoding on nodes in each level, we can traverse all nodes and for each node, check whether its corresponding block overlaps the query region via several efficient bit manipulations. We provide the details as follows.

Consider a node $N = (l, nid, isLeaf)$ in $\mathcal{I}_Q$. Since all the nodes in level l of the original quadtree virtually partition the whole region into $2^l \times 2^l$ virtual blocks, we denote the virtual block in the i-th row and j-th column by (i, j). There is an interesting relationship between (i, j) and nid, as follows.

Theorem 3: Given a virtual block (i, j) of node $N = (l, nid, isLeaf)$ in quadtree QT, the following equations hold:

$$i = \sum_{k \in [0,2l-1] \wedge (k\%2=1)} (\lfloor nid/2^k \rfloor \%2) \times 2^{\lfloor k/2 \rfloor}$$
$$j = \sum_{k \in [0,2l-1] \wedge (k\%2=0)} (\lfloor nid/2^k \rfloor \%2) \times 2^{\lfloor k/2 \rfloor} \quad (6)$$

which can be computed efficiently via bit manipulations.

By projecting the query region $R = (r_0, r_1)$ onto level-l virtual block unit coordinates, we can represent R as a pair $\{(i_0, j_0), (i_1, j_1)\}$ of virtual blocks, where (i_0, j_0) and (i_1, j_1) cover points r_0 and r_1, respectively. Hence, the virtual block (i, j) of N is overlapped with R iff $i \in [i_0, i_1] \wedge j \in [j_0, j_1]$.

Figure 3 shows an example for searching nodes of level 1 and level 2 in $\mathcal{I}_Q$. The query region is identified as the yellow dash line in level 1 and blue dash line in level 2. Level 1 involves $2^1 \times 2^1$ virtual blocks, where $(i_0, j_0) = (0,1)_2$ and $(i_1, j_1) = (1,1)_2$. Consider the nodes with nid of 1 and 3. Following Equation 6, we have $i = (0)_2, j = (1)_2$ and $i = (1)_2, j = (1)_2$, respectively. Hence, the blocks corresponding to these nodes overlap the query region. The block corresponding to the node $N = (1, 1, True)$ is inserted into $\mathcal{Q}_r^c$ as it is a leaf node, while the child nodes of node $N = (1, 3, False)$ are verified later. The same procedure is performed for nodes in level 2, in which the blocks corresponding to the nodes $N_1 = (2, 12, True)$ and $N_2 = (2, 13, True)$ are inserted into $\mathcal{Q}_r^c$.

After identifying all the blocks corresponding to the leaf nodes in $\mathcal{I}_Q$ that are overlapped with R, we extract points covered by each candidate block in $\mathcal{Q}_r^c$ from trajectory table $\mathcal{S}$. We then transfer these points to global memory of the GPU for the parallel verification. Note that points covered by a candidate block are stored continuously in $\mathcal{S}$ because of the Morton codes, and after transferring they are still stored continuously in global memory. To avoid redundant data transfers (i.e., overlap of candidate blocks among different queries), we maintain a bitmap table MAT recording the existence of each cell of points in global memory of the GPU. The total number of points in global memory is also recorded in the MAT, and the classical Least Recently Used (LRU) swapping strategy is used to swap some cells out of global memory if it is nearly full.

2) Verifying by GPU: The input to this step is a set $\mathcal{Q}_r^c$ of candidate blocks and the corresponding query region R of each candidate block. The output is a $|\mathcal{T}|$-wide bitmap $\mathcal{B}_r$ where the i-th bit equals to 1 iff trajectory $T_i \in \mathcal{Q}_r(R)$. $\mathcal{B}_r$ is initially set to 0s and after verification, it will be parsed by CPU to get the final result $\mathcal{Q}_r(R)$. The procedure of this step is shown in line 17-29 in Algorithm 2.

We assign points covered a candidate block to one CUDA block (handled by a GPU SM), which will then be distributed uniformly among CUDA threads. Since each candidate block corresponds to a leaf node in $\mathcal{I}_Q$, obtained by grouping size-varying cells, distributing candidate blocks among GPU SMs facilitates load balancing. In a CUDA block, for each retrieved point, a CUDA thread tries to check whether the point resides in R or not. If so, we set the corresponding bit in $\mathcal{B}_r$ to 1 (line 20-26). Note that in this step, data request from CUDA threads can be coalesced, because points in the same candidate block are stored continuously in global memory.

C. Top-k Similarity Query

Given a query trajectory T_q, the goal of top-k similarity query processing is to find the k trajectories closest to T_q using EDR as the distance function efficiently. In GAT, we consider a batch of similarity queries $\{T_q\}$ and execute them concurrently. Similar to range queries, we examine the MAT before transferring data from main memory to global memory, to eliminate redundant transfers among queries.

Algorithm 3: Top-k Similarity Query

```
Input: a set of query trajectories $\{T_q\}$, $\mathcal{S}$, $k$, $\epsilon$, $\mathcal{I}_T$;
Output: a result set of $k$-sized trajectory sets $\{\mathcal{Q}_k(T_q)\}$
1  for $T_q \in \{T_q\}$ parallelly do
2      $Q_{FD} \leftarrow$ initPriorityQueue("lowestOnTop");
3      $Q_k^r \leftarrow$ initPriorityQueue("highestOnTop", $k$);
4      // (1) computing frequency distance by CPU
5      for $T_i \in \mathcal{T}$ do
6          $FD(T_q, T_i) \leftarrow$ computeFD($FV(T_q, \mathcal{C})$, $FV(T_i, \mathcal{C})$);
7          $Q_{FD}$.push($T_i$, $FD(T_q, T_i)$);
8      end
9      $Terminate \leftarrow False$;
10     while $Terminate = False$ do
11         // (2) identifying candidate trajectories by CPU
12         $\mathcal{Q}_k^c \leftarrow Q_{FD}$.popTraj($m$);
13         transferTrajToGPU($\mathcal{Q}_k^c$, $\mathcal{I}_T$, $\mathcal{S}$);
14         // (3) EDR calculation by GPU
15         for CUDA block $bID \in [0, |\mathcal{Q}_k^c| - 1]$ parallelly do
16             $|T_q| \leftarrow$len($T_q$); $|T| \leftarrow$len($\mathcal{Q}_k^c[bID]$);
17             $S \leftarrow$ createArray(($|T_q| + 1) \times (|T| + 1)$);
18             for $i \in [1, (|T_q| + |T| - 1)]$ do
19                 for CUDA thread $thID$ parallelly do
20                     $sl_0, sl_1, sl_2 \leftarrow$getSlashOffset($S$, $i - 2$, $i$);
21                     $S[sl_0 + thID] \leftarrow$ computeState($S$, $sl_1$,
                         $sl_2$, $T_q$, $\mathcal{Q}_k^c[bID]$, $\epsilon$);
22                 end
23             end
24             $Q_k^r$.push($\mathcal{Q}_k^c[bID]$, $S[(|T_q| + 1) \times (|T| + 1) - 1]$);
25         end
26         if $(|Q_{FD}| = 0) \vee ((|Q_k^r| = k) \wedge (Q_{FD}.top().FD \geq$
             $Q_k^r.top().\delta))$ then
27             $Terminate \leftarrow True$;
28             break;
29         end
30     end
31     $\mathcal{Q}_k(T_q) \leftarrow Q_k^r$.popTraj($k$);
32 end
```

The computational cost of trajectory similarity queries comes from two aspects. First, the large number of trajectories in $\mathcal{T}$ makes it infeasible to compute EDR distance for each trajectory against T_q, i.e., $\delta(T_q, T), \forall T \in \mathcal{T}$. Second, the computational complexity of EDR is quadratic, i.e., $O(|T_q| \times |T|)$.

To avoid computing EDR distances for all trajectories, we leverage the lower bound of EDR distance, which is also known as *frequency distance* (FD) [12]. FD(T_q, T) is computed based on the *frequency vector*s (FV) of trajectories. In our context, we can consider each cell in $\mathcal{C}$ as a bin and the frequency vector of trajectory T over $\mathcal{C}$, denoted by FV($T, \mathcal{C}$), is a histogram recoding the number of points in T belonging to each cell. FD(T_q, T) is then defined as the minimum number of insert, delete, or replace EDR operations that are required to transform FV($T_q, \mathcal{C}$) to FV($T, \mathcal{C}$), or vice versa. Without loss of generality, we assume the side length of any cell exceeds the EDR threshold ϵ and allow adjacent histogram bins to be *approximately matched* as defined in [12]. We apply the same algorithm in [12] to compute FDs and the time complexity is linear to the number of bins in FVs.

Theorem 4: Given two trajectories T_q, T, and EDR threshold ϵ, we have FD(T_q, T)$\leq \delta(T_q, T)$ [12].

Inspired by Theorem 4, we propose to compute $\mathcal{Q}_k$ progressively. Given a query trajectory T_q and k, we first compute FD(T_q, T) for each trajectory $T \in \mathcal{T}$ and sort all the trajectories in ascending order of their FD values. We then iteratively choose $m(\geq k)$ trajectories with the smallest FDs as candidates $\mathcal{Q}_k^c$ and compute their EDR distances against T_q using the GPU. We keep k trajectories with the smallest EDR distances among all the distances computed so far. If all these k distances are no larger than FDs of the remaining trajectories (whose EDR distances have not been computed), the k trajectories are the top-k answers to $\mathcal{Q}_k$.

Corollary 1: For any subset $\mathcal{L}_\delta \subseteq \mathcal{T}$ of trajectories, all of them have smaller EDR distances than those in $\mathcal{T} - \mathcal{L}_\delta$ iff the maximum EDR distance of $\mathcal{L}_\delta$ is no larger than the minimum FD value of trajectories in $\mathcal{T} - \mathcal{L}_\delta$.

Algorithm 3 summarizes the pseudo code of executing similarity queries on the CPU-GPU environment in GAT. We maintain two priority queues: Q_{FD} contains the IDs of all trajectories in ascending order of FD(T_q, T), and Q_k^r includes k IDs for trajectories with computed EDR distances in ascending order of δ. Given $\mathcal{Q}_k$, our approach consists of three steps: (1) computing frequency distances for trajectories by CPU (line 4-8); (2) identifying a set $\mathcal{Q}_k^c$ of candidate trajectories by CPU (line 11-13); (3) performing EDR computation for candidates in $\mathcal{Q}_k^c$ against T_q by GPUs (line 14-29). Step 2 and 3 are performed iteratively until we get the complete query result $\mathcal{Q}_k(T_q)$ (line 9-31). We provide details of each step as follows.

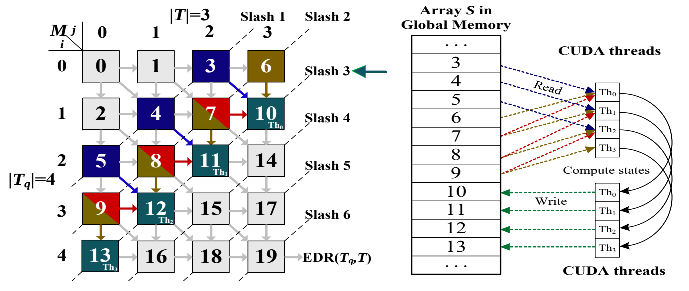
1) Computing Frequency Distances by CPU: The computation of FDs is based on FVs of trajectories. For each trajectory $T \in \mathcal{T}$, we compute its FV($T, \mathcal{C}$) based on the trajectory index $\mathcal{I}_T$. Recall that $\mathcal{I}_T$ maps a trajectory ID tid to a vector of $(cid, offset, length)$ triplets, i.e., L_{tid}. By traversing L_{tid} once, we can compute the total $length$s for each cid, which is exactly FV($T, \mathcal{C}$). Specifically, for each $C \in \mathcal{C}$, we have:

$$FV(T, \mathcal{C})[C] = \sum_{i=1}^{|L_{tid}|} \mathbb{1}_{L_{tid}.cid = C.cid} \times L_{tid}[i].length \tag{7}$$

Note that the FVs for trajectories in $\mathcal{T}$ can be computed and materialized before query processing. For query trajectory T_q, we first map each of its points to the covering cell according to the coordinates and then compute the number of points per cell to get FV($T_q, \mathcal{C}$). Then, for each trajectory $T \in \mathcal{T}$, we calculate the frequency distance FD(T_q, T) based on their frequency vectors. As the computational cost of FD(T_q, T) is linear to the number of bins in FVs [12], the overall complexity of this step is: O$((|\mathcal{T}| + 1)|\mathcal{C}|)$.

2) Generating candidate trajectories by CPU: We keep tids of the trajectories in $\mathcal{T}$ into a priority queue Q_{FD} ordered by FDs. In this step, we choose $m(\geq k)$ $tids$ from Q_{FD} with the smallest FD values and put the corresponding trajectories into $\mathcal{Q}_k^c$ as candidates. If Q_{FD} contains less than m tids, all these remaining trajectories become candidates. The tids of trajectories in $\mathcal{Q}_k^c$ are removed from Q_{FD} (line 12).

After generating $\mathcal{Q}_k^c$, we retrieve the points of trajectories whose IDs are in $\mathcal{Q}_k^c$ from $\mathcal{S}$ according to their cell-level representations stored in $\mathcal{I}_T$. We then transfer these trajectories



(a) Computing EDR on GPU in slash order (b) Data accessed by CUDA threads

Fig. 4. An example of EDR calculation on GPU

together with the query trajectory to global memory of the GPU for computing EDR (line 13). Similar to the range query, the MAT is also maintained to record the existence of a trajectory in global memory of the GPU, which the LRU swapping strategy is used to swap some trajectories out of global memory if it is nearly full.

3) Computing EDR distances by GPU: This step computes EDR distances for $\mathcal{Q}_k^c$. We use one CUDA block to compute the EDR distance between a candidate trajectory $T \in \mathcal{Q}_k^c$ and the query trajectory T_q, i.e., $\delta(T_q, T)$ (line 16-24). Without loss of generality, we assume $|T_q| \geq |T|$ in this section. For $|T_q| < |T|$, we can compute $\delta(T, T_q)$ as EDR is symmetric.

EDR distances can be computed through a *dynamic programming* prodedure. Let $\bar{T}(k)$ denote the subsequence of trajectory T from the k-th point to the end point. We maintain a $(|T_q| + 1) \times (|T| + 1)$ matrix M, where $M[i][j] = \delta(\bar{T}_q(i), \bar{T}(j))$. According to Definition 3, M satisfies:

$$M[i][j] = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min\{M[i-1][j] + 1, M[i][j-1] + 1, & \\ \quad M[i-1][j-1] + subcost\} & \text{otherwise} \end{cases} \tag{8}$$

The procedure of computing EDR can be divided into $(|T_q| + |T| - 1)$ iterations. We illustrate the idea using Figure 4(a), where each element in M is represented by a rectangle. We link all elements computed in i-th iteration with slash i. The computations of elements in slash i are independent of each other, and only depend on the elements in slash $i-1$ and $i-2$. Hence, we use one CUDA thread to compute an element in each slash. For example, we use CUDA threads $0, 1, 2, 3$ to compute $M[1][3], M[2][2], M[3][1], M[4][0]$, respectively. By doing this, the EDR distance $\delta(T_q, T)$ can be computed in $|T_q| + |T| - 1$ iterations. This is smaller than $|T_q| \times |T|$ iterations required by computing elements in a row major order.

In our implementation, we store all elements of M into an array S in global memory. To facilitate coalesced data access, we store elements in the same slash continuously in a *state array* S. Specifically, denoting $\alpha = (i + j - 1)$ and $\beta = (|T_q| + |T| - \alpha)$, $M[i][j]$ is stored in the $S[v]$, where:

$$v = \begin{cases} \frac{(\alpha+1)(\alpha+2)}{2} + i, & \text{if } \alpha \leq |T| - 1 \\ (|T| + 1)(|T_q| + 1) - \frac{(\beta+1)\beta}{2} + (|T| - j) & \text{if } \alpha \geq |T_q| - 1 \\ \frac{|T|(|T|+1)}{2} + (|T + 1|)(\alpha + 1 - |T|) + (|T| - j) & \text{otherwise} \end{cases} \tag{9}$$

Figure 4 (a) shows an example of the location of each state in array S, and from Figure 4 (b) we can see all the data accesses to global memory from CUDA threads 0, 1, 2, 3 are coalesced based on the data placement in array S.

After computing $\delta(T_q, T)$ for all $T \in \mathcal{Q}_k^c$, we put all the trajectory IDs to a k-sized priority queue Q_k^τ which maintains at most k trajectory IDs with the smallest EDR distances computed so far (line 24). If the maximum EDR distance is larger than the minimum FD in Q_{FD}, we continue to Step 2. Otherwise, the process terminates and outputs $\mathcal{Q}_k$ (line 26-31).

VI. Experimental Evaluation

A. Experiment Setup

Datasets. We use two real-life trajectory datasets: SHCAR and GeoLife [18]. SHCAR contains trajectories of 9,446 private cars in Shanghai, China, collected from 2014 to 2015. GeoLife contains trajectories from 182 users during Apr. 2007 – Aug. 2012, where most trajectory points are within Beijing, China. The original datasets contain one complete trajectory for each user. We split the sequence of trajectory points from each user into several subsequences, to obtain trajectories as $\mathcal{T}$. If two consecutive points have a time gap more than 30 minutes, we split them into two trajectories. Table I provides the details of two datasets.

Baselines. For range query, we compare GAT with its CPU version, and two GPU approaches: STIG [19] and FSG [20].

- **STIG [19].** This approach leverages kd-tree [21], where the leaf nodes correspond to blocks of points. We keep kd-tree in memory and use the CPU to compute candidate blocks, which will be refined by GPUs to get final results.
- **FSG [20].** This approach uses a flat grid-file based index to accelerate point-in-polygon queries over taxi trips. Similar to our approach, it first finds cells overlapped with the query region, then generates candidate cell-polygon pairs and sends them to the GPU for parallel verifications.
- **GAT-CPU-R.** This is the CPU version of GAT using multi-threading, where each candidate block is assigned to a thread running on the CPU for verification.

To the best of our knowledge, we are the first to exploit the GPU to accelerate top-k similarity query using EDR as the distance function. Hence, we only compare GAT with its multi-threaded CPU version.

- **GAT-CPU-S.** This is the CPU version of GAT, adapting histogram sequential scan [12] to multi-threaded processing.

Metrics. We measure the performance of the approaches by the total execution time over a batch of queries. For the indexing cost, we measure the construction time and the memory cost.

We run all the experiments on a server equipped with two 10-core Xeon E5-2650 v3 processor clocked at 2.3GHz, 64GB of RAM, 4TB of disk storage and a two-chip NVIDIA Tesla K80 GPU. Each chip has 2,496 CUDA cores and 12GB graphical memory. We implement GAT in C++ with CUDA 8.0, running on CentOS 7. We conduct our experiments under various parameter settings. Table II shows the meaning and the range of all parameters, where the default values are underlined. All the results are averaged over 10 runs.

TABLE I
Statistics of datasets

	#Traj.	avg. traj. length	#Points	#Cells
SHCAR	327,474	848	75,188,293	262,144
GeoLife	30,325	926.4	19,143,208	262,144

TABLE II
Parameter ranges and default values

Parameter	Description	Range
n	Max quadtree level	7, 8, <u>9</u>, 10
θ_p	Block size threshold	200, 2000, <u>20000</u>, 200000
S_R	Query region area (km^2)	10, <u>20</u>, 30, 40
k	Top-k for similarity query	5, 10, 15, <u>20</u>, 25
ζ	Avg. query trajectory length	200, 400, 600, <u>800</u>, 1000
$N_{\mathcal{Q}_r}$	#range queries	40, 60, <u>80</u>, 100, 120
$N_{\mathcal{Q}_k}$	#similarity queries	20, <u>40</u>, 60, 80, 100
$\lvert D\rvert$	dataset size (GB)	3, 4.5, 6, 7.5, <u>9</u>

B. Comparison of Various Approaches

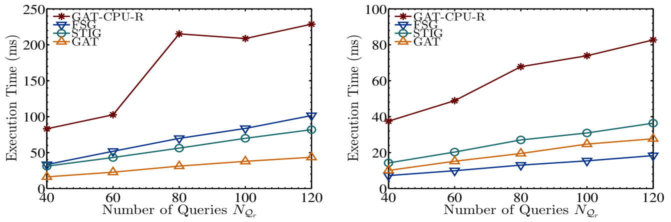
Range Query. Figure 5(a) and Figure 5(b) show the execution time with different numbers of range queries on SHCAR and Geolife, respectively. The execution time in all approaches increases when the number of queries becomes larger. Our approach outperforms the other three baselines in all cases and the advantage becomes more significant with more queries. On average, GAT runs 2.28x and 1.89x faster than FSG and STIG, respectively. We observe that GAT incurses lower data transfer cost due to the use of the memory allocation table, while the cost in FSG and STIG is relatively high. STIG performs better than FSG. This is because STIG uses a kd-tree like index to ensure similar number of trajectory points among blocks, leading to more balanced workload. This benefit is also achieved by GAT. GAT-CPU-R requires the longest execution time in all cases due to the limited multi-core parallelism.

Similarity Query. Figure 6(a) and Figure 6(b) show the execution time with different number of similarity queries on SHCAR and Geolife, respectively. GAT runs up to 4.5x faster than GAT-CPU-S over all the query numbers, indicating higher throughput of our approach using GPU. While the number of GPU cores is 100x more than that of CPU cores, the throughput can be compromised by various factors such as the different clock rates of CPU and GPU, and the high latency of PCIe bus for data transfer. On two datasets, the execution time of GAT and GAT-CPU-S increases linearly as the number of queries becomes larger. Both approaches require longer execution time on SHCAR which contains more trajectories.

C. Indexing Cost

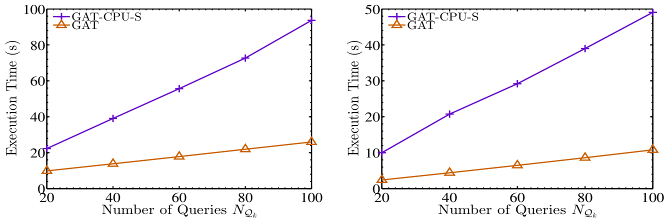
We evaluate the indexing cost for three GPU approaches. We randomly select the subsets of trajectories from both SHCAR and GeoLife, and build indices for each subset.

Memory Cost. Figure 7(a) shows the index memory cost w.r.t. different data sizes. For all the approaches, the size of indices becomes larger when the data size increases. GAT requires slightly more memory cost (8%-22%) compared with other two methods. The main reason is that GAT involves



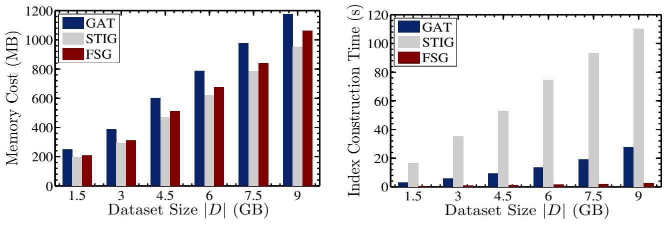
(a) SHCAR (a) GeoLife

Fig. 5. Comparison of performance on range queries



(b) SHCAR (b) GeoLife

Fig. 6. Comparison of performance on similarity queries



(a) Memory Cost (b) Index Construction Time

Fig. 7. Indexing cost

trajectory index to perform histogram-based pruning for similarity queries, while STIG and FSG can only process range queries and do not maintain such an index. On average, the overhead of GTIDX is merely 12.8% of the data size, which is acceptable and space-efficient.

Index Construction Time. Figure 7(b) provides the index construction time for different data sizes. The construction time increases as the data size becomes larger. For 9GB data, GAT takes only 27.7 seconds for the index construction, which is efficient. GAT requires 75.2% less index construction time than STIG over all the data sizes. This is because STIG needs to compute a large number of medium values when generating kd-tree. The index construction time of GAT is longer than that of FSG due to the additional tree index $\mathcal{I}_Q$ in GAT.

D. Speedup

We next study the speedup achieved by GAT with default parameter values on one and two GPUs. For comparison, we use a single thread to execute all the queries on CPU. The speedup is computed as the ratio of execution time against the single-thread approach. Table III shows the results on two datasets. For range queries, GAT with one GPU achieves 20.07x and 18.03x speedups on SHCAR and GeoLife, respectively, owning to the high parallelism of GPU. Using two GPUs achieves about 1.76x speedup than using one GPU over both datasets. This indicates that GAT can achieve almost linear speedup with more GPUs. For similarity queries,

TABLE III
SPEEDUP ACHIEVED ON TWO DATASETS

Dataset	**SHCAR**		**GeoLife**	
#GPUs	**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

GAT with one GPU achieves 35.54x and 38.94x speedups on SHCAR and GeoLife, respectively. The speedups are higher than those for range queries. Compared with similarity queries, range query processing involves simple comparison operations and the data transfer cost between main memory and global memory may become more significant in the overall execution time, which restricts the increase of speedup.

E. Scalability

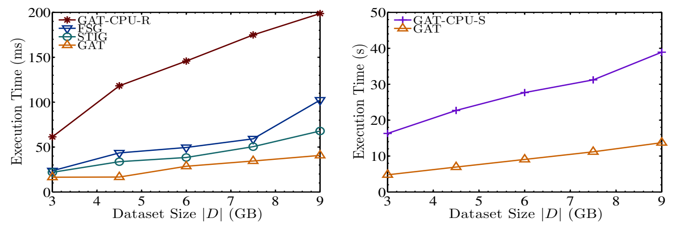
We investigate the scalability of different approaches by varying the data size from 3GB to 9GB. Figure 8(a) shows the execution time of 80 range queries over different data sizes. All the approaches require longer execution time when the dataset becomes larger. This is determined by the fact that large datasets typically involve more candidates to be verified. GAT scales better than FSG, as GAT groups cells into blocks to achieve load balancing. This phenomenon is more significant on larger datasets, which limits the scalability of FSG. Figure 8(b) reports the execution time of similarity queries over different data sizes. As data volume grows, both GAT and GAT-CPU-S require longer execution time. However, the execution time of GAT grows much slower than that of GAT-CPU-S, indicating better scalability using the GPU.

F. Parameter Tuning

In this section, we study the effects of different parameter values on the performance of GAT, as listed in Table II.

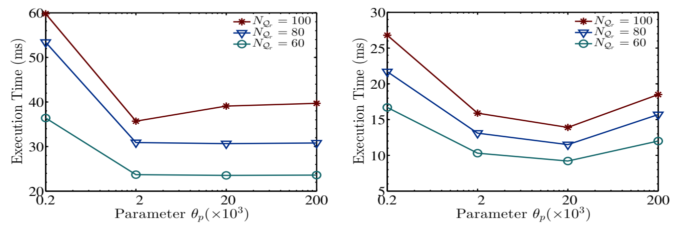
Effect of query region area S_R (km^2). The area of query region S_R controls the selectivity of range query. All the query regions are generated by extending from a point in the city center. Figure 9 shows the results with different S_R. The execution time of all approaches increases when the area of R varies from $10km^2$ to $40km^2$. As the query region becomes larger, more blocks overlap with R, leading to more candidates for verification. GAT performs the best over all the areas on GeoLife, but runs slightly slower than FSG for $S_R = 10km^2$ on SHCAR. The reason may be that SHCAR involves a small number of points residing in a small query region, but GAT verifies the whole block if any of its cells is overlapped with R. The GPU solutions outperform GAT-CPU-R in all cases, due to the superior of the GPU in parallel computations.

Effect of block size threshold θ_p. Figure 10 shows the execution time of range queries with various values of θ_p. Recall that a large value of θ_p means that more cells are grouped into a large block, while the total number of blocks is reduced. For both datasets, we observe that the execution time of GAT decreases significantly when θ_p varies from 200 to 2000. When θ_p is too small, the number of points per block becomes smaller and verifying a few points in a candidate block cannot fully utilize the parallelism of GPU SM. We



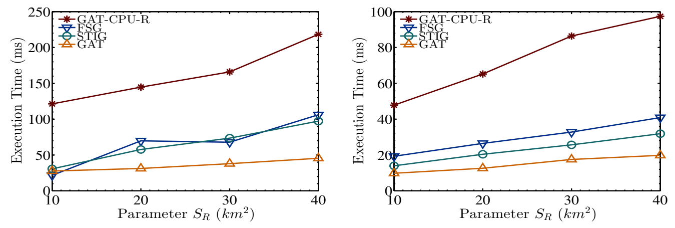
(a) Range Query (b) Similarity Query

Fig. 8. Scalability



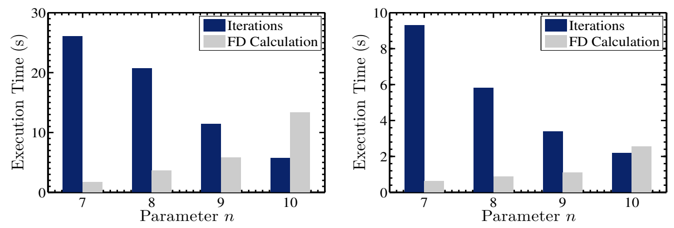
(a) SHCAR (b) GeoLife

Fig. 10. Effect of block size threshold (θ_p)



(a) SHCAR (b) GeoLife

Fig. 9. Effect of query region area (S_R)



(a) SHCAR (b) GeoLife

Fig. 11. Effect of maximum quadtree level (n)

also observe a slight increase in execution time when θ_p varies from $20,000$ to $200,000$. This is because larger θ_p results in more points per block and the size difference among blocks becomes larger. Such variance causes unbalanced workloads.

Effect of maximum quadtree level n. The value of n controls the number of cells in $\mathcal{C}$. As we use θ_p to group cells into blocks, the performance of GAT on range queries is quite stable with different n. We thus omit the results due to the space limit. Figure 11 provides the execution time of similarity queries on two datasets by varying n. As n increases, calculating frequency distance requires more execution time. The reason is that the length of FV is exponential to n and the computational cost of FD is linear to the length of FV. The time of the iterations for generating candidates and calculating EDR decreases as n becomes larger, because more cells lead to tighter lower bounds of EDR and stronger pruning power.

Effect of k. Figure 12 shows the execution time of similarity queries under different values of k. As k becomes larger, both GAT and GAT-CPU-S require longer execution time. This is because larger k requires more EDR distances to be computed. However, for GAT, the increase in the execution time is small as the EDR computations are performed by thousands of GPU cores and the extra cost with larger k can be amortized.

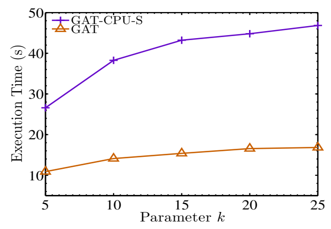
Effect of average query trajectory length ζ. We study the effect of average query trajectory length on similarity query processing. For each length, we randomly choose 50 trajectories and report the averaged result. Figure 13 shows the results. For both SHCAR and GeoLife, GAT outperforms GAT-CPU-S over all the trajectory lengths. The reason is that GAT computes each EDR distance with one SM and the computational cost is proportional to the sum of two trajectory lengths in our proposed algorithm. However, GAT-CPU-S adopts the original dynamic programming method and the complexity of computing EDR distance is proportional to the multiplication of two trajectory lengths. The advantage of GAT thus becomes more significant for larger ζ.

VII. Related Work

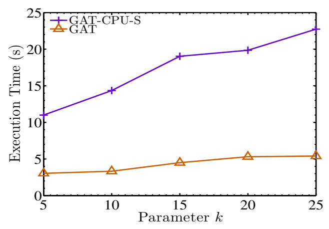
We review related works divided into two categories: trajectory indexing and GPU-based trajectory query processing.

Trajectory indexing. R-Tree [22] is one of most classical indices for spatial data, which is a two-dimensional generalization of B-Tree [23]. After that, some indices such as 3D-RTree [24], TB-Tree [25] and TPR-Tree [26] have been proposed to index trajectory data. SETI [17] proposed a two-level indexing scheme by decoupling spatial and temporal dimensions for spatial queries. PIST [27] developed a cost model to measure the IO cost of different partitioning over spatial-temporal data, in order to reduce the number of disk accesses. TrajStore [16] proposed an adaptive griding scheme for indexing, clustering and storing trajectory data. Wang et al. developed an in-memory column-oriented storage called SharkDB [8] for trajectories, where range and similarity queries can be executed in memory efficiently. Trajtree [11] was designed to index the computation of EDwP, a similarity metric for trajectories under inconsistent sampling rates. All these indexing schemes are designed for spatial query processing on CPU, which can be incorporated into the filtering part of our framework. However, they are not optimized for GPU-accelerated verification, thus inspiring us to develop GTIDX.

GPU-based trajectory query processing. The GPU has been used for accelerating query processing over trajectory data. Zhang et al. [28] proposed a system called U$_2$STRA to manage large-scale trajectory data efficiently on GPU. They proposed to split trajectories with a four-level hierarchy and index them by a grid-file based data structure. Based on this system, an algorithm called TKSimGPU [9] was developed for top-k similarity queries. The idea is to compute similarity for pairs of trajectories using GPU. However, this approach adopted Hausdorff distance as the similarity measure, which is not robust to noise in real trajectories. And the solution cannot support more accurate similarity metrics such as EDR distance [12]. Lettich et al. [15] proposed the PR-quadtree

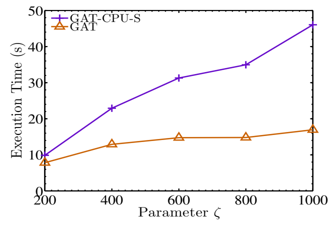


(a) SHCAR

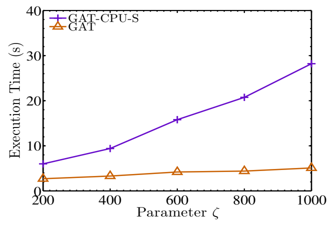


(b) GeoLife

Fig. 12. Effect of k



(a) SHCAR



(b) GeoLife

Fig. 13. Effect of query trajectory length (ζ)

to process stream spatial k-nearest neighbor queries, where a PR-quadtree index was developed to achieve load balancing in query processing. However, they require to load all the data into global memory for processing, which cannot scale to large datasets. Zhang et al. [20] developed a query processing system with a grid-file based data structure to manage taxi trips. Doraiswamy et al. [19] generalized the kd-tree to STIG in order to support iterative spatial-temporal queries using the GPU. However, these approach does not support trajectory similarity queries and the performance on range queries is inferior to GAT.

VIII. Conclusion

This paper has presented GAT, a GPU-accelerated framework to support two types of trajectory queries together (i.e., both range and similarity queries) over trajectory data. GAT follows the generic filtering-and-verification framework, where the candidates are computed on the CPU, and the massive parallel verifications are performed on the GPU. To accelerate query processing on the GPU, we introduce a space-efficient index to effectively filter invalid trajectories for both types of trajectory queries. The Morton-based encoding is applied to permit data access requests from the GPU cores to be coalesced, which addresses the limited global memory bandwidth problem. To achieve load balancing among GPU SMs, we group size-varying cells into balanced blocks with similar numbers of trajectory points. Extensive experiments were conducted using two real-life trajectory datasets. The results demonstrate that GAT achieves significant performance advantages over existing CPU-based solutions, and runs about 2x faster than the state-of-the-art GPU-based methods that support only one type of queries (i.e., range queries).

References

[1] M. Kosoff. (2015, Jun.) Uber is getting dwarfed by its chinese rival, which gives 3 million rides a day. [Online]. Available: http://www.businessinsider.com/chinas-uber-rival-didi-kuaidi-does-3-million-rides-a-day-2015-6

[2] G. Hu, J. Shao, F. Shen, Z. Huang, and H. T. Shen, "Unifying multi-source social media data for personalized travel route planning," in *SIGIR*, 2017, pp. 893–896.

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

[4] W.-C. Lee, W. Si, L.-J. Chen, and M.-C. Chen, "HTTP: a new framework for bus travel time prediction based on historical trajectories," in *ACM GIS*, 2013, pp. 279–288.

[5] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: a high-performance, distributed main memory transaction processing system," in *PVLDB*, vol. 1, no. 2, 2008, pp. 1496–1499.

[6] 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 *SIGMOD*, 2015, pp. 1383–1394.

[7] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *SIGMOD*, 2016, pp. 1071–1085.

[8] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, and S. W. Sadiq, "Sharkdb: An in-memory column-oriented trajectory storage," in *CIKM*, 2014, pp. 1409–1418.

[9] E. Leal, L. Gruenwald, J. Zhang, and S. You, "TKSimGPU: A parallel top-k trajectory similarity query processing algorithm for GPGPUs," in *Big Data*, 2015, pp. 461–469.

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

[11] S. Ranu, P. Deepak, A. D. Telang, P. Deshpande, and S. Raghavan, "Indexing and matching trajectories under inconsistent sampling rates," in *ICDE*, 2015, pp. 999–1010.

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

[13] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

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

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

[16] P. Cudré-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *ICDE*, 2010, pp. 109–120.

[17] V. P. Chakka, A. Everspaugh, and J. M. Patel, "Indexing large trajectory data sets with SETI," in *CIDR*, 2003.

[18] 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.

[19] 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 *ICDE*, 2016, pp. 1086–1097.

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

[21] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[22] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[23] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," in *SIGFIDET Workshop*, 1970, pp. 107–141.

[24] Y. Theodoridis, M. Vazirgiannis, and T. K. Sellis, "Spatio-temporal indexing for large multimedia applications," in *ICMCS*, 1996, pp. 441–448.

[25] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches to the indexing of moving object trajectories," in *VLDB*, 2000, pp. 395–406.

[26] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López, "Indexing the positions of continuously moving objects," in *SIGMOD*, 2000, pp. 331–342.

[27] 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.

[28] J. Zhang, S. You, and L. Gruenwald, "U2stra: High-performance data management of ubiquitous urban sensing trajectories on GPGPUs," in *CDMW*, 2012, pp. 5–12.