A GPU Accelerated Update Efficient Index for kNN Queries in Road Networks

Chuanwen Li #, Yu Gu #, Jianzhong Qi *, Jiayuan He *, Qingxu Deng #, Ge Yu #

# *College of Computer Science and Engineering, Northeastern University, China*
{lichuanwen, guyu, dengqingxu, yuge}@mail.neu.edu.cn

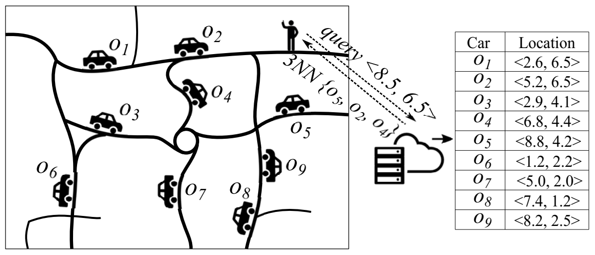
* *School of Computing and Information Systems, The University of Melbourne, Australia*
jianzhong.qi@unimelb.edu.au, hjhe@student.unimelb.edu.au

***Abstract*—The k nearest neighbor (kNN) query in road networks is a traditional query type in spatial databases. This query has found new applications in the fast-growing location-based services, e.g., finding the k nearest Uber cars of a user for ride-sharing. KNN queries in these applications are non-trivial to process due to the frequent location updates of data objects (e.g., movements of the cars). This calls for novel spatial indexes with high efficiency in not only query processing but also update handling. To address this need, we propose an index structure that uses a "lazy update" strategy to reduce the costs of update handling without sacrificing query efficiency or answer accuracy. We cache the location updates of data objects and only update the corresponding entries in the index when they are queried. We further propose a kNN query algorithm based on this index. This algorithm takes advantage of the strengths of both the CPU and the GPU. It first identifies the queried region and updates the index over this region using the GPU. Then, it uses the GPU to query the index and produce a candidate result set, which is later refined by the CPU to obtain the final query answer. We conduct experiments on real data and compare the proposed algorithm with state-of-the-art kNN algorithms. The experimental results show that the proposed algorithm outperforms the baseline algorithms by orders of magnitude in query time.**

I. Introduction

Location-based services (LBS) have become increasingly popular with the prevalence of smart mobile devices. On-going efforts are made to improve the user experience of mobile LBS through improvements on query processing efficiency over moving objects [1], [2], [3]. We study a basic type of LBS queries, the k *nearest neighbor* (kNN) query, over moving objects in road networks [1]. The kNN query reports the k nearest neighbors of a given query object. We consider data objects with location changes, i.e., moving data objects. Note that even though moving data objects are considered, we compute the kNN only based on a snapshot of the locations of the data objects at query time. We do not compute the query answer continuously as the data objects move, i.e., we are not studying continuous kNN queries.

An example of such a query is to find the three nearest Uber cars for a user, where the cars are moving and the nearest cars are computed based on their distances to the query object at query time. Figure 1 illustrates such a query. Nine cars, denoted as $o_1, o_2, ..., o_9$, are moving on a road network. Their locations are stored in a query server. A user issues a query containing her current location, e.g., $\langle 8.5, 6.5\rangle$, to the server to request three nearest cars. The server retrieves the



Car	Location
o_1	<2.6, 6.5>
o_2	<5.2, 6.5>
o_3	<2.9, 4.1>
o_4	<6.8, 4.4>
o_5	<8.8, 4.2>
o_6	<1.2, 2.2>
o_7	<5.0, 2.0>
o_8	<7.4, 1.2>
o_9	<8.2, 2.5>

Fig. 1: A kNN query example

current locations of the cars, computes the three nearest cars, e.g., $\langle o_5, o_2, o_4\rangle$, and sends them back to the user. The main challenge to address such a query is that the server needs to solicit constant updates from the cars to keep their locations up-to-date at the time of query processing.

Existing techniques for kNN queries over static data objects are not directly applicable for moving objects. Many of these techniques rely on index structures fine-tuned for static data objects. When applied to moving objects, the costs to maintain such index structures may become too high to be practical.

The state-of-the-art road network kNN algorithm, *V-tree* [4], allows dynamic object updates. It uses a balanced search tree to index moving objects and precomputed tables to facilitate network shortest path distance computation. Each object update triggers an index update. This can be expensive when the update frequency is high.

We propose an index structure for kNN queries in road networks with high update efficiency. We make use of an observation that object location updates only need to be enforced on the index structure when the relevant objects are queried. This allows us to simply cache a location update upon receiving it instead of actually enforcing the update on the index. At query time, we flush the cached updates into the index before searching for the query answer. We design a highly parallel algorithm using the GPU to update the index and hence ensure a high update efficiency.

We design a kNN algorithm based on the proposed index. This algorithm combines the strength of both the CPU and the GPU. Upon receiving a query, it first uses the CPU to identify the queried region and then uses the GPU to process the cached updates in this region. To compute the query answer, it follows a filter-and-refine scheme. It uses the GPU to compute a candidate answer set and the CPU to compute a final answer.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00084

GPUs have hundreds or thousands of cores that can run in parallel. Such a feature suits nicely the tasks of (lazily) processing the object location updates and finding the kNNs at query time. In particular, the location updates of different objects are all independent from each other, which can be easily parallelized on a GPU. There are also challenges, especillay in processing location updates of the same object. Finding the latest location of an object with multiple cached location updates may incur synchronization issues, since GPU threads need to read the location updates, compare them with the latest location of the object found so far, and (potentially) update the latest location of the object in parallel. We address the synchronization issues via taking advantage of the *butterfly shuffle* functionality offered by GPUs that allows swapping data between GPU threads with a low cost (detailed in Section IV-C2). Once the latest locations of the data objects are identified, we find the kNNs by using the GPU to compute the distances between the query object and data objects in parallel and the CPU to compare the distances computed. This way, we take advantage of both the parallel power of the GPU and the high efficiency in comparing and branching of the CPU.

In summary, we make the following contributions:

- We propose a novel index structure and an index updating algorithm that caches and delays index updates until the indexed entries to be updated are queried, to reduce the index update frequency.
- We propose a highly parallel algorithm on the GPU for cleaning cached updates simultaneously. The algorithm adopts a low-cost shuffle strategy to solve the synchronization issues and achieve a high concurrency.
- We propose a CPU-GPU collaborating kNN query processing algorithm. The algorithm takes advantage of the high parallel power of the GPU for distance computations of a large number of data objects and the high efficiency of the CPU in branching operations to compare the distances and produce the final query answer.
- We perform cost analysis and extensive experiments. The results confirm the superiority of the proposed algorithms over the state-of-the-art algorithms.

The remainder of the paper is organized as follows. We present preliminaries in Section II. We describe the proposed index structure in Section III. We detail the index update algorithms in Section IV and the kNN query algorithm in Section V. We report cost analysis and experimental results in Section VI and Section VII. We review related studies in Section VIII and conclude the paper in Section IX.

II. Preliminaries

We consider a road network represented as a directed graph $G = \langle V, E\rangle$, where V is a set of vertices and E is a set of edges. An edge $\overrightarrow{e_{ij}} \in E$ connects two vertices v_i and v_j. We call v_i and v_j the source and the destination vertices of $\overrightarrow{e_{ij}}$. The weight of $\overrightarrow{e_{ij}}$, denoted as $\overrightarrow{e_{ij}}.w$, represents the cost for traveling from v_i to v_j. Note that this model can be extended to undirected graphs easily by replacing an undirected edge with two directed edges of the same weight. In what follows, for simplicity, we use e instead of $\overrightarrow{e}$ to represent an edge when the edge direction is irrelevant in the context.

TABLE I: Frequently used symbols

Notation	Description
G	a directed graph
v	a vertex
$\overrightarrow{e_{ij}}$	an edge from vertex v_i to v_j
c	a cell of G
o	a moving data object
m	a message of an object location update
ζ	a message bucket
$c.l$	the message list of cell c
$\mathcal{S}$	the set of messages read in by a bundle of threads

We assume a static query object o_q and a set of n data objects $\mathcal{O} = \{o_1, o_2, \ldots, o_n\}$, of which the locations may change over time.

Definition 1 (KNN query): Given a road network G, a query object o_q, a set of data objects $\mathcal{O}$ moving on G, and an integer query parameter k, a kNN query over the set $\mathcal{O}$ returns a size-k subset $\mathcal{R} \subseteq \mathcal{O}$ such that, at the time when the query is issued, $\forall o_i \in \mathcal{R}, o_j \in \mathcal{O}\backslash\mathcal{R} : \texttt{dist}(o_q, o_i) \leqslant \texttt{dist}(o_q, o_j)$.

Here, function $\texttt{dist}(o_i, o_j)$ returns the network distance (length of the graph shortest path) from o_i to o_j.

We use a server that hosts an object location index to process kNN queries. Each data object reports its location updates to the server periodically. The time interval between two location updates must not exceed a predefined value $t_\triangle$. This interval determines how far away the actual kNNs could be from the kNNs computed at query time. A smaller $t_\triangle$ produces more accurate results but also brings a higher update workload. The value of $t_\triangle$ is constrained by the processing power of the server. We consider it as a given system parameter.

We call each update a *message* and denote it by m, $m = \langle o, e, d, t\rangle$. Here, $m.o$ denotes the object that sends the message, $m.e$ denotes the edge on which $m.o$ locates, d denotes the distance from the source vertex of $m.e$ to $m.o$, and $m.t$ denotes the update time.

The query server needs to process the messages in time to produce accurate query answers. The workload for message processing is proportional to $\sum_i^n f_i$, where f_i is the update frequency of object o_i. When the number of objects is large or the update frequency is high, a non-trivial update workload will be incurred on the server.

III. G-Grid Index

We propose an index structure, named *G-Grid*, for processing kNN queries over data objects with location updates in road networks. The G-Grid index consists of three parts: a *graph grid* that represents the road network (detailed in Section III-A), an *object table* that indexes the locations of objects (detailed in Section III-B), and a set of *message lists* that cache the object location updates (detailed in Seection III-C).

A. Graph Grid

We use a graph grid, which is a grid based structure, to index a road network graph. Each cell c in a graph grid stores a set of vertices. Each vertex v is associated with a set of edges where v is the destination vertex. We maintain two identical graph grids in the memory of both the CPU and the GPU. To reserve memory locality for highly parallel accesses on the GPU, our graph grid is based on arrays instead of pointer-based hierarchical structures. Next, we detail how to build a graph grid for a given road network graph G.

Cells. Given a graph $G = \langle V, E \rangle$ and an integer δ^c, we map the vertices in V into $2^\psi \times 2^\psi$ cells where $\psi = \lceil \frac{1}{2} \log_2 \frac{|V|}{\delta^c} \rceil$. Each cell contains at most δ^c vertices. Here, we call δ^c the *cell capacity*. It controls the maximum number of vertices in a cell. We adopt the graph partitioning algorithm by Karypis and Kumar [5] to partition the graph. This graph partitioning algorithm iteratively divides a set of vertices into equal-sized subsets while minimizing the number of edges between vertices in two subsets. The two subsets form two neighboring cells, which are further partitioned into smaller cells. After the graph partitioning, we represent a cell c with a 3-tuple: $c = \langle \mathcal{A}_v, n_v, n_e \rangle$. Here, $c.\mathcal{A}_v$ is a size-δ^c array storing all the vertices in c; $c.n_v$ is the number of vertices in $c.\mathcal{A}_v$; and $c.n_e$ is the number of edges of which the source vertices are in c.

We store the cells in a one-dimensional array according to the *Z-curve* [6]. For each cell c, we map its two-dimensional coordinate (x, y), i.e., its position in the grid, to its *Z-value* z, which is used as its position in the one-dimensional array. Here, the Z-value of c is computed by interleaving the binary representations of y and x. For example, a cell with a coordinate (3,4) has a Z-value of $(37)_{10} = (100101)_2$, which is the result of interleaving $(4)_{10} = (100)_2$ and $(3)_{10} = (011)_2$. This mapping transfers the two dimensional grid graph to a one dimension array while preserves data locality. Nearby cells in the grid graph often co-locate in the array. Data locality is crucial to the performance of the GPU algorithm.

Vertices. Each element in $c.\mathcal{A}_v$ represents a vertex v that locates in c, $v = \langle id, \mathcal{A}_e, n \rangle$, where $v.id$ is the vertex ID, $v.\mathcal{A}_e$ is a size-δ^v array that stores the edges having v as the destination vertex, and $v.n$ is the number of edges stored in $v.\mathcal{A}_e$. Array $v.\mathcal{A}_e$ may not be full, and $v.n$ may not be δ^v. Here, we use a parameter δ^v to control the number of edges stored with a vertex. We call this parameter the *vertex capacity*. If vertex v is the destination vertex of more than δ^v edges, we create *virtual vertices* for v. A virtual vertex is represented as $v' = \langle id', \mathcal{A}'_e, n \rangle$, and is stored in the same cell c. We add edges beyond the capacity limit to these virtual vertices. The number of virtual vertices for a vertex v is $\lceil \frac{v_{ne}}{\delta^v} \rceil$, where v_{ne} is the number of edges having v as the destination vertices.

Edges. Each element in $v.\mathcal{A}_e$ represents an edge e of which the destination vertex is v, $e = \langle id, v_s, w \rangle$, where $e.id$ is the edge ID, $e.v_s$ is the source vertex, and $e.w$ is the edge weight.

Inverted index. Together with the grid, we maintain an inverted index (a hash table) that maps an edge to the IDs of its source vertex and the cell where this vertex locates.

Figure 2 illustrates a graph grid. We partition a network graph into 64 cells and map the cells into an array using a Z-curve (the pink curve). We use the superscript to represent the position of the cell in the array, e.g., c^0 is the first element in the array. Assume that $\delta^c = 5$ and $\delta^v = 10$. Each cell uses a size-5 array for storing its vertices and each vertex uses a size-10 array for storing its edges. The cell c^{21}, for example, contains an array $\mathcal{A}_v^{21}$ where each element represents a vertex in c^{21}. For the first vertex v^0, its array $\mathcal{A}_e^0$ stores the edges that have v^0 as the destination vertex.

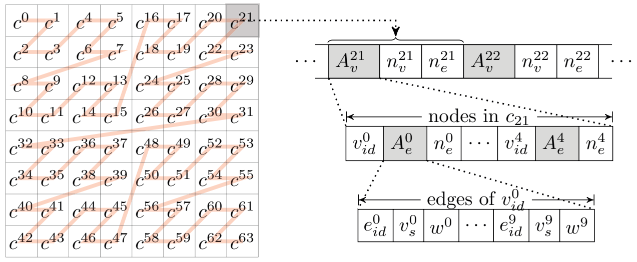

c^0 c^1 c^4 c^5 c^{16} c^{17} c^{20} c^{21}
c^2 c^3 c^6 c^7 c^{18} c^{19} c^{22} c^{23}
c^8 c^9 c^{12} c^{13} c^{24} c^{25} c^{28} c^{29}
c^{10} c^{11} c^{14} c^{15} c^{26} c^{27} c^{30} c^{31}
c^{32} c^{33} c^{36} c^{37} c^{48} c^{49} c^{52} c^{53}
c^{34} c^{35} c^{38} c^{39} c^{50} c^{51} c^{54} c^{55}
c^{40} c^{41} c^{44} c^{45} c^{56} c^{57} c^{60} c^{61}
c^{42} c^{43} c^{46} c^{47} c^{58} c^{59} c^{62} c^{63}
$\cdots$ A_v^{21} n_v^{21} n_e^{21} A_v^{22} n_v^{22} n_e^{22} $\cdots$
nodes in c_{21}
v_{id}^0 A_e^0 n_e^0 $\cdots$ v_{id}^4 A_e^4 n_e^4
edges of v_{id}^0
e_{id}^0 v_s^0 w^0 $\cdots$ e_{id}^9 v_s^9 w^9

Fig. 2: A graph grid

B. Object Table

We use a hash table named the *object table* to store the latest locations of all objects. Each entry of the table is a key-value pair: $o.id \mapsto \langle c.id, e.id, d \rangle$. Here, $o.id$ is the ID of an object o, $c.id$ is the ID of the cell that contains the latest location of o, $e.id$ is the ID of the edge e that contains the latest location of o, and d is the distance from the source vertex of e to o. We store the object table in the CPU memory.

C. Message Lists

We use *message lists* to store messages sent from data objects. We maintain a message list for each graph grid cell, where the messages are stored in their chronological order. The message list puts every δ^b messages into a bucket and use a linked list to index the created buckets. Here, δ^b represents the maximum number of messages allowed in a bucket, i.e., the *bucket capacity*. A message list L maintains three pointers p_h, p_t, and p_l. Pointers p_h and p_t point to the head and the tail buckets, respectively. Pointer p_l points to a bucket ζ_i, before which the buckets are locked (detailed in Section IV-C).

A bucket ζ is a 4-tuple: $\zeta = \langle \mathcal{A}_m, n, t, p_n \rangle$, where $\zeta.\mathcal{A}_m$ is a size-δ^b array for storing messages, $\zeta.n$ is the number of messages stored in $\zeta.\mathcal{A}_m$, $\zeta.t$ is the time of the latest message in ζ, and $\zeta.p_n$ is a pointer that points to the next bucket.

We store message lists in main memory and send them to the GPU for processing when needed, as detailed in Section IV-C.

IV. Updating the G-Grid

The query server needs to update the G-Grid to keep the object locations up-to-date. Existing studies update their index structures when there is an object location update [4], [7]. This "eager" approach may have unnecessary updates which are overwritten by new updates before any query is issued. We take a "lazy" approach instead. When a message arrives, we cache it in a message list and only process it when its corresponding cells are queried. This enables skipping the updates that do not impact query answers. We further propose a GPU-based algorithm to clean up the unnecessary messages before performing updates on the G-Grid. Next, we detail how to cache a message and clean up cached messages.

A. Caching a Message

When a new message m is received, we first append m to its corresponding cell. Then, we update the object table. This procedure is summarized as Algorithm 1. The algorithm first calls getCell() to identify the cell c to which m belongs. This function takes the edge, i.e., $m.e$, and uses the inverted index in the graph grid to identify cell c. Then, the algorithm inserts m to the last bucket in the message list of c, denoted as $c.l$. If the last bucket is full, the algorithm creates a new bucket containing m and appends the bucket to $c.l$. If the updated object $m.o$ has moved to cell c from another cell c', in addition to the above procedure, the algorithm also adds another message m' to cell c'.

Algorithm 1: Updating the G-Grid

```
input : a message $m$ and the G-Grid
output: updated G-Grid
1 $c$ = getCell($m.e$)
2 append($c.l, m$)
3 if getCellFromOT($m.o$) $\neq c$ then
4     $c'$ = getCellFromOT($m.o$)
5     append($c'.l, \langle m.o$, null, null, $m.t\rangle$)
6 setOT($m.o, \langle c, m.e, m.d\rangle$)
```

B. Message Cleaning

A message list of a cell records recent object location updates in the cell. To locate objects in a cell, we retrieve the messages cached in its message list and remove the obsolete messages. We call this procedure *message cleaning*.

Message cleaning can be costly when the number of cached messages is large. To improve the cleaning efficiency, we take advantage of the parallel processing capability of the GPU. Given a set $\mathcal{L}$ of message lists, we process $\mathcal{L}$ in four steps: (1) preprocessing; (2) GPU memory preparation; (3) parallel processing; and (4) result computation. These four steps are summarized in Algorithm 2 and are detailed next. Here, "$\lll \ggg$" represents a function that runs in parallel on the GPU. The integer parameter enclosed by $\lll\ggg$ represents the number of threads.

1) Preprocessing: In this step, we lock all the message lists in $\mathcal{L}$ and copy the valid buckets over to the GPU.

Let L be a message list in $\mathcal{L}$. We create a new empty bucket ζ_{new} and append it to the tail of L. We let pointer p_l pointing to ζ_{new} (Lines 1-3). This operation helps identify whether L is under processing. Anytime before we send L to the GPU for clean-up, we compare its pointers p_l and p_h. If the two pointers are pointing to different buckets, we can skip L safely.

For each bucket ζ before p_l, we check if the time of its latest message, i.e., $\zeta.t$, satisfies $\zeta.t - t_{now} < t_{\triangle}$. Here, t_{now} represents the time of message cleaning. If yes, we keep the bucket in L. Otherwise, we discard the bucket. Recall that each object needs to send at least one update message in $t_{\triangle}$ time. If the latest message in ζ is sent before $t_{now} - t_{\triangle}$, all messages in ζ must be outdated and can be discarded.

We copy the remaining buckets to the GPU. Assuming that the number of remaining buckets in $\mathcal{L}$ is $\mathcal{L}.n$, we create a size-$\mathcal{L}.n$ array, denoted as $\mathcal{L}.A$, for these buckets (Line 4). In $\mathcal{L}.A$, we attach each message with an ID of the cell to which the message belongs, i.e., each message is now represented as a 5-tuple: $m = \langle o, c, e, d, t\rangle$.

Algorithm 2: Message_Cleaning

```
input : a set of message lists $\mathcal{L}$
output: up-to-date object locations $\mathcal{R}$
1 for each $L \in \mathcal{L}$ parallel do
2     if $p_l \neq p_h$ then skip $L$
3     append a new bucket $\zeta$ to the tail of $L$; point $p_l$ to $\zeta$
4 create a size $\mathcal{L}.n$ array $\mathcal{L}.A$ on the GPU
5 copy buckets before $p_l$ for all $L \in \mathcal{L}$ into $\mathcal{L}.A$
6 create $\mathcal{T}$ and $\mathcal{R}$ on the GPU
7 $\mathcal{T} \leftarrow$ GPU_X_Shuffle$\lll\mathcal{L}.n\ggg$ ($\mathcal{L}.A, \eta$)
8 $\mathcal{R} \leftarrow$ GPU_Collect $\lll|\mathcal{T}|\ggg$ ($\mathcal{T}$)
9 return $\mathcal{R}$
```

2) Preparing Memory on the GPU: We create two hash tables $\mathcal{T}$ and $\mathcal{R}$ to store the intermediate and final results of message cleaning, respectively (Line 6).

The *intermediate result table* $\mathcal{T}$ stores candidate locations of the objects. Each entry in $\mathcal{T}$ is a key-value pair, where the key is an object o and the value is an array storing the candidate messages of o, which will be detailed in Section IV-C.

The *final result table* $\mathcal{R}$ stores the final and up-to-date locations of the objects. The key of an entry in $\mathcal{R}$ is the ID of a cell c. The value of an entry is a set of objects in c.

For simplicity, we use $\mathcal{T}[o]$ and $\mathcal{R}[c]$ to represent the values in $\mathcal{T}$ and $\mathcal{R}$ of a given key o and c, respectively.

3) Parallel Processing: We allocate a thread in the GPU for each bucket in $\mathcal{L}.A$. Every thread processes the messages in its assigned bucket sequentially. Each time, all the threads read in a message from their assigned buckets simultaneously. To process a message m sent by an object o, a thread retrieves from $\mathcal{T}[o]$ the latest message of o found so far, denoted as m'. If m is newer than m', the thread replaces m' with m.

This approach has a synchronization issue. When multiple threads are processing messages of the same object, these threads will need to access the same entry in $\mathcal{T}$. We will detail how to solve this synchronization issue in Section IV-C.

4) Result Computation: We fill table $\mathcal{R}$ based on $\mathcal{T}$. For each entry $\mathcal{T}[o]$, we create a thread. The thread performs a GPU_Collect procedure to retrieve the latest message m_{new} of o remaining in $\mathcal{T}$ after the parallel processing step. Then, we insert an entry of o into $\mathcal{R}$, and set its key is $m_{new}.c$. We send the completed table $\mathcal{R}$ back to CPU, where we use $\mathcal{R}$ to update the message lists of the corresponding cells.

C. Parallel Message Processing

Message processing in parallel using the GPU faces has a synchronization issue where several threads need to access the same entry in $\mathcal{T}$. A straightforward approach to address this issue is to lock $\mathcal{T}[o]$ once a thread has gained access to it. Other threads can only access the entry when the lock is released. However, this approach increases the waiting time among threads and hence degrades the efficiency.

We propose a lock-free algorithm for message processing.

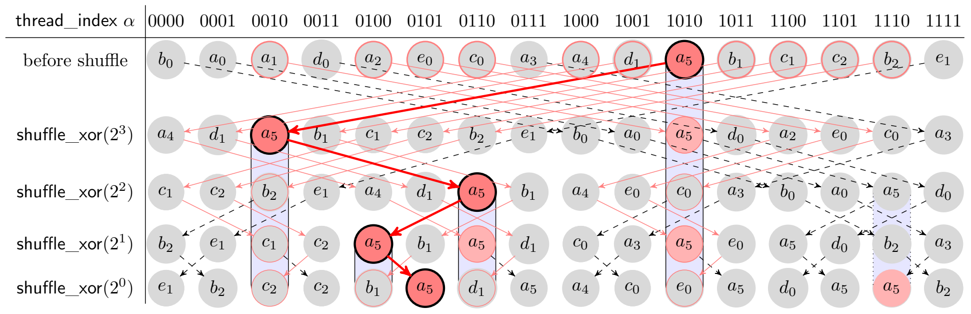

thread_index α
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
before shuffle
b_0 a_0 a_1 d_0 a_2 e_0 c_0 a_3 a_4 d_1 a_5 b_1 c_1 c_2 b_2 e_1
shuffle_xor(2^3)
a_4 d_1 a_5 b_1 c_1 c_2 b_2 e_1 b_0 a_0 a_5 d_0 a_2 e_0 c_0 a_3
shuffle_xor(2^2)
c_1 c_2 b_2 e_1 a_4 d_1 a_5 b_1 a_4 e_0 c_0 a_3 b_0 a_0 a_5 d_0
shuffle_xor(2^1)
b_2 e_1 c_1 c_2 a_5 b_1 a_5 d_1 c_0 a_3 a_5 e_0 a_5 d_0 b_2 a_3
shuffle_xor(2^0)
e_1 b_2 c_2 c_2 b_1 a_5 d_1 a_5 a_4 c_0 e_0 a_5 d_0 a_5 a_5 b_2

Fig. 3: X_shuffle on a bundle of 16 threads

1) Thread Bundles: We divide all the threads into equi-sized bundles, each with 2^η threads. These bundles are numbered from 0 to $n-1$, where $n = \lceil \frac{\mathcal{L}.n}{2^\eta} \rceil$. Recall that in table $\mathcal{T}$, we have created an array for each data object. This array stores the candidate messages for each data object found in these n bundles. Specifically, we set the size of each array as n. Given an object o, we store the newest message of o found by the i-th bundle as the i-th element $\mathcal{T}[o][i]$ in $\mathcal{T}[o]$. Now we only need to synchronize threads of the same bundle, which is done by the `GPU_X_Shuffle` method described as follows.

Algorithm 3: `GPU_X_Shuffle`

```
input : a bucket $\mathcal{L}.A[\textbf{\textit{thread\_id}}]$, an integer $\eta$, a differential factor $\mu$,
        and a hash table $\mathcal{T}$
output: the updated hash table $\mathcal{T}$
1  create a size-$\eta$ array $\Gamma$
2  bundle_id $\leftarrow$ thread_id$/2^\eta$
3  for $i = \delta^m - 1$ to 0 do
4      $m \leftarrow \mathcal{L}.A[$thread_id$][i]$
5      for $j = 1$ to $\eta$ do
6          $m_\Gamma \leftarrow \Gamma$.getMessage$(m.o)$
7          if $m_\Gamma =$ null then add $m$ to $\Gamma$
8          else if $m_\Gamma.t < m.t$ then replace $m_\Gamma$ by $m$ in $\Gamma$
9          else $m \leftarrow m_\Gamma$
10         $m \leftarrow$ shuffle_xor$(m, 2^{\eta-j})$
11     repeat $\mu(\eta)$ times
12         if $\mathcal{T}[m.o][$bundle_id$].t < m.t$ then
13             $\mathcal{T}[m.o][$bundle_id$] \leftarrow m$
```

2) X-Shuffle: We use `GPU_X_Shuffle` to solve the intra-bundle synchronization issue with the following two steps:

Step 1. Given a bundle of 2^η threads and a size-2^η set $\mathcal{S}$ of messages read in simultaneously by these threads, we keep the latest message of each object while reducing the number of messages of the same object in $\mathcal{S}$ to a small integer k, where k is bounded by $\mu(\eta)$. Here, $\mu(\eta)$ is a function that maps η to an integer much smaller than 2^η (detailed in Section IV-D).

Step 2. For each object o in $\mathcal{S}$, we repeat update $\mathcal{T}[o][i]$ for $\mu(\eta)$ times, where i is the bundle number (Lines 12 to 14).

Step 1 is done using a special group of GPU functions named *warp shuffle*. The warp shuffle functions allow active threads within a thread group to exchange data at very low costs. We utilize the warp shuffle function that performs *butterfly shuffle* among the threads. This function makes each thread to exchange the data with another thread, of which the thread id is the bitwise xor result between the current thread id and a parameter s, i.e., threads j and $j \oplus s$ will exchange data. We call s the *lane mask* in a butterfly shuffle. For example, given a bundle of 4 threads, a `shuffle_xor`(2) exchanges data between the 0^{th} and 2^{nd}, the 1^{st} and 3^{th} threads since $(00)_2 \oplus (10)_2 = (10)_2$ and $(01)_2 \oplus (10)_2 = (11)_2$.

We make use of the butterfly shuffle and design a "cache-and-shuffle" algorithm that can safely and efficiently reduce the number of messages of the same object in $\mathcal{S}$. We summarize the cache-and-shuffle algorithm for each thread in a bundle of threads in Lines 1 to 11 of Algorithm 3. The algorithm creates a size-η array Γ for each thread, which is used as a message cache. Then, the algorithm performs butterfly shuffles for η times (Lines 5 to 10). The lane mask of the j-th shuffle is set as $2^{\eta-j}$. After each shuffle, every thread compares the message shuffled to it with messages in its message cache. Assume that a message m has been shuffled to a thread. This thread first finds the latest message of the same data object as that of m, i.e., $m.o$, in its message cache Γ. If no message of $m.o$ is found or the found message, denoted as m_Γ, is outdated compared with m, the algorithm updates Γ by replacing m_Γ with m. Otherwise, the algorithm discards m and updates its message to be shuffled as m_Γ.

The above procedure guarantees that the number of messages of the same object in $\mathcal{S}$ will not exceed $\mu(\eta)$. At the end of this procedure, we only need to further check messages of the same object for at most $\mu(\eta)$ times to find the latest message of each object in each bundle in $\mathcal{T}$ (Lines 11 to 13).

We further illustrate `G_X_Shuffle` using Fig. 3. Assume a bundle of 16 threads indexed from $(0000)_2$ to $(1111)_2$. When `G_X_Shuffle` starts, each thread reads a message as shown in the row "before shuffle". Here, we use a to e to represent messages sent by five different data objects and the subscript to represent the chronological order of the messages. For example, a_1 and a_2 are both messages of object a while a_1 is an earlier message compared with a_2. Each thread first adds its message to its message cache. For example, the message cache of thread $(0010)_2$ is now $[a_1]$. Next, the algorithm performs a butterfly shuffle with a lane mask of $2^3 = (1000)_2$. Each thread compares its new message with its message cache and replaces outdated messages. The result is shown in the row "`shuffle_xor`(2^3)". For example, message a_5 is shuffled to thread $(0010)_2$. This thread looks for messages of a in its cache and finds a_1. Since a_5 is newer, the thread replaces a_1 by a_5 in the cache. As another example, message a_1 is shuffled to thread (1010_2). Since a_1 is older compared with the cached message a_5 of thread (1010_2), the thread keeps its cache as $[a_5]$ and uses a_5 for the next shuffle. As a third example, thread $(0000)_2$ with a cache of $[b_0]$ gets a message a_4 in the shuffling. This thread updates its cache as $[a_4, b_0]$. We repeat the above procedure for 3 more iteration except that the lane mask value decreases by a factor of 2 for each iteration. The resultant messages of the threads are shown in the row "`shuffle_xor(2^0)`". At the end, the number of messages of the same object is no more than $\mu(4) = 2$ (e.g., there are only two messages of a: a_4 and a_5).

D. Upper Bound of the Number of Messages of an Object

Function $\mu(\eta)$ represents the maximum number of different messages of an object in a bundle after message shuffling. This parameter is critical to the algorithm performance since it determines the number of updates required in $\mathcal{T}$. In this subsection, we show that $\mu(\eta)$ has a very small value compared with the number of threads, i.e., 2^η. To give an example, assume a bundle of 32 threads. The maximum number of different messages of an object will be reduced to 4 after message shuffling. Since we need to perform η butterfly shuffles, each thread only needs to process $\eta+1 = 5$ messages. When updating the table $\mathcal{T}$, each thread only needs to write the table $\mu(\eta) = 4$ times to guarantee that the latest message of an object is successfully written in $\mathcal{T}$. Compared to the straightforward approach which needs to write $\mathcal{T}$ for $\eta = 32$ times, now the total number of operations is reduced to 9.

Theorem 1: Given a bundle of 2^η threads where $\eta > 3$, the number of messages of an object is bounded by $\mu(\eta)$ after the shuffles, where

$$\mu(\eta) = \begin{cases} \arg\min_i \lambda(\eta, i) \geq 2^\eta & \lambda(\eta, 8) \geq 2^\eta \\ 2^\eta - \lambda(\eta, 8) + 8 & \text{otherwise} \end{cases} \tag{1}$$

Here, $\lambda(\eta, i) = i\binom{\eta+1}{2} - \sum_{j=1}^{i} \frac{(14-j)(j-1)}{2} + i$.

For bundles where the total numbers of threads is $2^\eta = 16, 32, 64, 128, \ldots$, the corresponding $\mu(\eta) = 2, 4, 8, 16, \ldots$.

In the following, we prove the correctness of Theorem 1. At the start of `GPU_X_Shuffle`, each thread reads a message. Then, `GPU_X_Shuffle` shuffles messages between the threads. In what follows, we use m_α to represent a message that is initially read in by thread α. Given two threads α and β, we say that α covers β, if m_β will be shuffled to a thread that m_α has previously appeared. Suppose that m_α and m_β are of the same object and $m_\alpha.t > m_\beta.t$. Then, it is straightforward that m_β will be replaced by m_α if and only if α covers β. Thus, given a bundle of messages, the maximum number of different messages of an object remained after shuffling is equal to the number of threads that cannot cover each other in the bundle. We call such a set of threads as an *exclusive set*. Next, we derive the maximum number of threads that a thread can cover. After that, we derive the maximum size of an exclusive set.

Theorem 2: Given a size-2^η thread bundle, let m_α be the message read in by thread α before shuffling. Message m_α will be shuffled to thread $\alpha \oplus \sum_{i=1}^{k} 2^{\eta-k}$ at the k^{th} shuffle if it is not replaced in the previous $k-1$ shuffles, where $k \in [1, \eta]$.

Proof: At i^{th} shuffle, the lane mask is set as $2^{\eta-i}$. Thus, after the k^{th} shuffle, m_α is shuffled to the thread $\alpha \oplus 2^{\eta-1} \oplus 2^{\eta-2} \oplus \cdots \oplus 2^{\eta-k}$. It is straightforward that $2^{\eta-1} \oplus 2^{\eta-2} \oplus \cdots \oplus 2^{\eta-k} = \sum_{i=1}^{k} 2^{\eta-k}$. Thus, the theorem is proven. ■

Definition 2 (X-distance): Given two threads α and β, the *x-distance* of α and β, $\mathcal{X}(\alpha, \beta)$, is the number of sequences result from splitting $\alpha \oplus \beta$ with 0.

For example, $\mathcal{X}(10, 1) = 2$, since $(01010)_2 \oplus (00001)_2 = (01011)_2$. We call a sequence an *order-x sequence* if splitting the sequence with 0 results in x sequences. For example, $(01011)_2$ is an order-2 sequence.

Lemma 1: Given two threads α and β, α covers β if and only if $\mathcal{X}(\alpha, \beta) = 1$.

Proof: (1) Since α covers β, m_β will be shuffled to one thread after m_α has been shuffled to the same thread. Let θ be the thread. Thus, we have: (i) $\theta = \alpha \oplus \sum_{i=0}^{j} 2^{\eta-i}$; (ii) $\theta = \beta \oplus \sum_{i=1}^{k} 2^{\eta-i}$; and (iii) $0 \leqslant j < k \leqslant \eta$. Note that $j = 0$ means m_α has not been shuffled yet. Thus, we have $\alpha \oplus \sum_{i=1}^{j} 2^{\eta-i} = \beta \oplus \sum_{i=1}^{k} 2^{\eta-i}$. Let both sides perform xor with the sequence $(\beta \oplus \sum_{i=1}^{j} 2^{\eta-i})$, then we have $\alpha \oplus \beta = \sum_{i=1}^{j} 2^{\eta-i} \oplus \sum_{i=1}^{k} 2^{\eta-i} = \sum_{i=j+1}^{k} 2^{\eta-i}$. The binary form of $\sum_{i=j+1}^{k} 2^{\eta-i}$ is a sequence of all 0 except the $(j+1)^{th}$ to k^{th} bits. Thus, we have $\mathcal{X}(\alpha, \beta) = 1$.

(2) Let $\gamma = \alpha \oplus \beta$. If $\mathcal{X}(\alpha, \beta) = 1$, γ is an order-1 sequence. Thus, there is one and only a continuous subsequence of 1 in the binary form of γ. Assume there are total η digits of γ, the first 1 is at the j^{th} bit and the last 1 at the k^{th} bit where $1 \leqslant j \leqslant k \leqslant \eta$. Then, the binary form of γ can be written as $\sum_{i=j}^{k} 2^{\eta-i}$. Combine the equation with $\alpha = \beta \oplus \gamma$, we have $\alpha = \beta \oplus \sum_{i=j}^{k} 2^{\eta-i}$. Let both sides perform xor with the sequence $\sum_{i=0}^{j-1} 2^{\eta-i}$, we have $\alpha \oplus \sum_{i=0}^{j-1} 2^{\eta-i} = \beta \oplus \sum_{i=j}^{k} 2^{\eta-i} \oplus \sum_{i=0}^{j-1} 2^{\eta-i} = \beta \oplus \sum_{i=0}^{k} 2^{\eta-i}$. Since $\alpha \oplus \sum_{i=0}^{j-1} 2^{\eta-i}$ represents the thread index of m_α at $(j-1)^{th}$ shuffle, $\beta \oplus \sum_{i=0}^{k} 2^{\eta-i}$ represents the thread index of m_β at k^{th} shuffle, and $j-1 < k$, m_α must cover m_β.

Combining (1) and (2), the theorem is proven. ■

Lemma 2: Given a size-2^η bundle of threads, let $\mathcal{C}(\alpha)$ be the set of threads covered by α, we have $|\mathcal{C}(\alpha)| = \binom{\eta+1}{2}$.

Proof: Let β be a thread from $\mathcal{C}(\alpha)$. Then, $\alpha \oplus \beta$ is an order-1 sequence, implying that its binary form is all 0 except the digits from j to k. Here, j and k are integers where $1 \leqslant j \leqslant k \leqslant \eta$. Since $\alpha \oplus \beta$ has η bits, there are $\binom{\eta+1}{2}$ possible combinations of j and k values. ■

Lemma 3: Given a size-2^η bundle of threads ($\eta > 3$), let α and β be two threads from the bundle. The number of messages covered by both m_α and m_β satisfies that

$$|\mathcal{C}(\alpha) \cap \mathcal{C}(\beta)| = \begin{cases} 6 & \mathcal{X}(\alpha, \beta) = 2 \\ 0 & \mathcal{X}(\alpha, \beta) > 2 \end{cases} \tag{2}$$

Proof: Let ψ be a thread covered by both α and β. From Lemma 1, we have $\mathcal{X}(\psi, \alpha) = 1$. We can write $\psi \oplus \alpha$ as $\sum_{i=x_1}^{x_2} 2^{\eta-i}$ where x_1 and x_2 represent the positions of the first and the last 1's in the result sequence of $\psi \oplus \alpha$. Similarly, we write $\psi \oplus \alpha$ as $\sum_{i=x_3}^{x_4} 2^{\eta-i}$.

Case 1, $\mathcal{X}(\alpha, \beta) = 2$: From Definition 2, we know that $\alpha \oplus \beta$ contains two subsequences of 1. Let y_1 and y_2 be the positions of the first and the last 1's in the first subsequence, and y_3 and y_4 be the positions of the first and the last 1's in the second subsequence, we can write $\alpha \oplus \beta$ as $\sum_{i=y_1}^{y_2} 2^{\eta-i} \oplus \sum_{i=y_3}^{y_4} 2^{\eta-i}$, where $1 \leqslant y_1 \leqslant y_2 < y_3 \leqslant y_4 \leqslant \eta$. Since $\alpha \oplus \beta = (\psi \oplus \alpha) \oplus (\psi \oplus \beta)$, we get $\sum_{i=x_1}^{x_2} 2^{\eta-i} \oplus \sum_{i=x_3}^{x_4} 2^{\eta-i} = \sum_{i=y_1}^{y_2} 2^{\eta-i} \oplus \sum_{i=y_3}^{y_4} 2^{\eta-i}$. It is straightforward that the set $\{x_1, x_2, x_3, x_4\}$ is a one-to-one mapping of the set $\{y_1, y_2, y_3, y_4\}$. Combined with the requirements that $x_1 \leqslant x_2$ and $x_3 \leqslant x_4$, there are six possible ways of mapping. Hence, $|\mathcal{C}(\alpha) \cap \mathcal{C}(\beta)| = 6$.

Case 2, $\mathcal{X}(\alpha, \beta) > 2$: The sequences $\psi \oplus \alpha$ and $\psi \oplus \beta$ are both order-1 sequences. Thus, $\alpha \oplus \beta = (\psi \oplus \alpha) \oplus (\psi \oplus \beta)$ is at most an order-2 sequence. This contradicts to $\mathcal{X}(\alpha, \beta) > 2$. Therefore, no such ψ exists and $|\mathcal{C}(\alpha) \cap \mathcal{C}(\beta)| = 0$.

Combining Cases (1) and (2), the theorem is proven. ■

Lemma 4: Given a size-2^η bundle ($\eta > 3$), let $\{\alpha, \beta, \gamma\}$ be an exclusive set of the bundle. We have

$$|\bigcap_{i \in \{\alpha,\beta,\gamma\}} \mathcal{C}(i)| = \begin{cases} 1 & \mathcal{X}(\alpha, \beta) = \mathcal{X}(\beta, \gamma) = \mathcal{X}(\alpha, \gamma) = 2 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Proof: Case 2 is straightforward from *Case 2* of Lemma 3. We focus on *Case 1*. Let $\psi_1 = \alpha \oplus \beta$ and $\psi_2 = \alpha \oplus \gamma$. Thus, ψ_1 and ψ_2 are both order-2 sequences. Further, $\psi_1 \oplus \psi_2 = \beta \oplus \gamma$, implying that $\psi_1 \oplus \psi_2$ is also an order-2 sequence. Let $\psi' = \psi_1 \wedge \psi_2$. We use $p(x)$ to represent a function that takes a sequence x and returns a new sequence $x \oplus x'$. Here, x' is the sequence of shifting x from the right to the left by 1 bit, concatenating it by 0, and removing the first bit. For example, $p((110001)_2) = (010011)_2$. Intuitively, each "1" in $p(x)$ corresponds to a position where the bit value changes in x. For an order-2 sequence x, the number of 1's in $p(x)$ should be 4. Next, we prove the number of 1's in $p(\psi_1) \wedge p(\psi_2)$ to be 2. Assume that the number of 1's in $p(\psi_1) \wedge p(\psi_2)$ is larger than 2. Then, the positions where ψ_1 and ψ_2 change bit values have at least 3 same positions. Since ψ_1 and ψ_2 are order-2 sequences, i.e., the number of positions where the bit values change does not exceed 4, $\psi_1 \oplus \psi_2$ is at most an order-1 sequence. Similarly, if the number of 1's in $p(\psi_1) \wedge p(\psi_2)$ is less than 2, $\psi_1 \oplus \psi_2$ is at least an order-3 sequence. Thus, the number of 1's in $p(\psi_1) \wedge p(\psi_2)$ is 2. Let ψ' be the number that satisfies $p(\psi') = p(\psi_1) \wedge p(\psi_2)$, e.g., $\psi_1 = (010100)_2, \psi_2 = (100010)_2, \psi' = (011100)_2$. We have $\mathcal{X}(\alpha \oplus \psi', \alpha) = \mathcal{X}(\alpha \oplus \psi', \beta) = \mathcal{X}(\alpha \oplus \psi', \gamma) = 1$, and $\mathcal{C}(\alpha) \cap \mathcal{C}(\beta) \cap \mathcal{C}(\gamma) = \{\alpha \oplus \psi'\}$. Based on the construction of ψ', it must be the only one that satisfies the above equation.

Combining Case (1) and (2), the theorem is proven. ■

Lemma 5: Given a size-2^η bundle and an exclusive set Λ of the bundle where $|\Lambda| \leqslant 8$. We have:

$$|\bigcup_{\alpha \in \Lambda} \mathcal{C}(\alpha)| \geq \min(\lambda(\eta, |\Lambda|),\ 2^\eta), \tag{4}$$

where $\lambda()$ is defined in Theorem 1.

Proof: We first discuss the case where the pairwise x-distances between any two threads in Λ is 2. For ease of discussion, we call such a set as *step-2 set*. It is straightforward that a step-2 set is a special case of an exclusive set. We will first show that the cardinality of a step-2 set is at most 8. Let α and β be two threads in Λ. Then for each thread γ from the set $\Lambda \setminus \{\alpha, \beta\}$, the set formed by α, β, and γ covers 1 and only 1 thread (Lemma 4). Since the total number of threads covered by both α and β is 6 (Lemma 3), the cardinality of the set $\Lambda \setminus \{\alpha, \beta\}$ is at most 6. Thus, the cardinality of Λ is at most 8. Suppose a step-2 set Λ covers n threads and $|\Lambda| = k$. Given another thread α that is 2 in x-distance to each thread in Λ, the set Λ' formed by $\Lambda \cup \{\alpha\}$ is also step-2 set. The thread α covers $\binom{\eta+1}{2}$ threads. However, the number of threads covered by Λ' is smaller than $n + \binom{\eta+1}{2}$ since some threads covered by α is also covered by threads in Λ. Given a thread β in Λ, α and β cover 6 six same threads (Lemma 3). Given two threads β and γ in Λ, the set $\{\alpha, \beta, \gamma\}$ covers 1 same thread (Lemma 4). Thus, when the size of Λ increase by 1, the number of its covered threads increases $\binom{\eta+1}{2} + \binom{k}{2} - 6k$. Thus, Equation 4 holds true for step-2 sets.

If Λ is not a step-2 set, let α and β be two threads satisfying $\mathcal{X}(\alpha, \beta) > 2$. Then, the number of threads covered by both α and β is 0 other than 6 in the case where $\mathcal{X}(\alpha, \beta) = 2$ (Lemma 3). Compared to a step-2 set of the same size as Λ, the total number of threads covered will be more. ■

Now we can prove Theorem 1:

Case 1, $\lambda(\eta, 8) \geqslant 2^\eta$: The function $\Lambda(\eta, i)$ increases monotonically with i. Let i be the smallest value that satisfies $\lambda(\eta, i) \geq 2^\eta$. From Lemma 5, we know that a size-i exclusive set is enough to cover all the 2^η threads.

Case 2, $\lambda(\eta, 8) < 2^\eta$: The size-8 exclusive set covers $\lambda(\eta, 8)$ threads. There are at most $2^\eta - \lambda(\eta, 8)$ covered threads.

V. *k*NN Query Processing

Straightforwardly, we can send all the data objects to the GPU and use it to compute a *k*NN answer. However, since transferring data to the GPU is expensive, processing *k*NN queries solely on the GPU is often not the best. We take advantages of both the GPU and the CPU. Specifically, we first use the GPU to compute a candidate result set. Then, we use the CPU to refine the candidate set and obtain the query answer. We present our *k*NN algorithm in Algorithm 4. The algorithm consists of three steps: (1) identify the cells that may contain the *k*NN answer and send their message lists to the GPU for message cleaning (Lines 1 to 4); (2) construct a candidate result set (Lines 5 to 9); and (3) refine the candidate set to obtain a final answer (Lines 10).

A. Selecting Candidate Objects

We first select the candidate cells, i.e., the cells that may contain the query answer. This can be done by computing the distance between the query object and cells, which will be detailed later in this subsection. For the selected cells, we send their message lists to the GPU for message cleaning.

Transferring message lists. Data transfer between the CPU and the GPU is relatively slow compared with message processing. To save time, we use a pipelined strategy, i.e., let the

GPU process and receive messages simultaneously. We group the message lists into several sets. When the first set arrives, the GPU starts to process the message lists immediately. In the meantime, it keeps receiving more message lists transferred from the CPU. The splitting procedure is straightforward and is omitted in Algorithm 4.

Algorithm 4: *k*NN Query Processing

```
input : query $q = \langle k, e, d\rangle$, $\rho$
output: result set $\mathcal{R}_q$
1  $c_q \leftarrow$ getCell($q.e$), $\mathcal{L} \leftarrow \{c_q \cup$ getNeighbors($\{c_q\}$)$\}$
2  $\mathcal{C} \leftarrow$ Message_Cleaning($\mathcal{L}$)
3  while $|\mathcal{C}| < \rho \cdot q.k$ do
4      $\mathcal{C} = \mathcal{C} \cup$ Message_Cleaning(neighbors($\mathcal{L}$) $\setminus \mathcal{L}$)
5  $\mathcal{V} \leftarrow$ getVertices($\mathcal{C}$)
6  $\mathcal{M} \leftarrow$ getObjects($\mathcal{C}$)
7  GPU_SDist $\lll|\mathcal{V}|\ggg$ ($\mathcal{V}$)
8  $\mathcal{R}_{can} \leftarrow$ GPU_First_k $\lll|\mathcal{M}|\ggg$ ($\mathcal{M}$)
9  $\mathcal{U} \leftarrow$ GPU_Unresolved $\lll|\mathcal{V}|\ggg$ ($\mathcal{V}$)
10 $R_q \leftarrow$ Refine_kNN($\mathcal{R}_{can}$, $\mathcal{U}$)
```

Selecting candidate cells. We compute a set of candidate cells that contain more than $\rho \cdot k$ objects. We adopt an iterative procedure to select the cells. Here, k is the query parameter and $\rho > 1$ is a system parameter for workload balancing between the GPU and the CPU. Let cell c_q be the cell of the query object, the CPU first sends c_q and its neighboring cells to the GPU. A cell c_i is a neighbor of another cell c_j if there exists an edge with a source vertex in c_i and a destination vertex in c_j. Let $\mathcal{C}$ be the set of objects found in the processed cells. We repeatedly send the neighboring cells of previously selected cells to the GPU until $|\mathcal{C}| \geqslant \rho \cdot k$ (Lines 3 to 4).

Parameter $\rho > 1$ balances the workloads of the CPU and the GPU. A larger value of ρ will increase the GPU workload and reduce the CPU workload. We will discuss the impact of ρ on the query performance in Section VI-B2 and perform empirical studies in Section VII.

B. Constructing a Candidate Result Set

In this step, we construct candidate results. Current *k*NN algorithms for road networks usually rely on Dijkstra's algorithm to compute the shortest path distance between the query and the data objects [4]. However, Dijkstra's algorithm computes the shortest path distance by iterating over the vertices, where each iteration is dependent on the result of the previous iteration. It is difficult to run Dijkstra's algorithm in parallel. We propose a different approach to achieve parallel processing so as to make full use of the GPU. Our algorithm consists of two steps: (1) compute the candidate objects; and (2) construct the *unresolved vertex set*.

Computing candidate objects. We first compute the shortest path distance from q to each vertex in the candidate cells, denoted as $\mathcal{V}$. We adapt the Bell-ford algorithm for parallel processing on the GPU, as summarized in Algorithm 5 GPU_SDist(). The algorithm computes the shortest distance to each vertex by repeatedly relaxing all the edges connected to vertices in $\mathcal{V}$ for $|\mathcal{V}|$ times. Since we have stored the edges with the same destination vertex together in the G-Grid, we can relax edges with different destination vertices in parallel without any conflicts in data accessing. We use $|\mathcal{V}|$ threads and assign each thread with a vertex v (Line 3). In each iteration, each thread relaxes the edges stored in v, i.e., $v.\mathcal{A}_v$, and updates the shortest path distance of v (Lines 5 to 6).

We compute the distance of q to data objects in $\mathcal{C}$ based on the shortest path distances of the vertices. Given a data object o, its distance from q is $\texttt{dist}(q,o) = D[m.e.v_s] + m.d$, where m is the latest message from o. Then, the candidate set is computed as the k data objects with smaller distances from q than any other data objects. In Algorithm 4, function GPU_First_k represents the above procedure (Line 8).

Algorithm 5: GPU_SDist

```
input : a set vertices $\mathcal{V}$
output: an array $D$ where each vertex $v$ is associated with its shortest
        distance to $q$
1  repeat $|\mathcal{V}|$ times
2      for $i \leftarrow 0$ to $\delta^{v} - 1$ do
3          $v \leftarrow \mathcal{V}$[thread_id]
4          $e \leftarrow v.\mathcal{A}_e[i]$
5          if $D[v] > D[e.v_s] + e.w$ then
6              $D[v] = D[e.v_s] + e.w$
7          sync_threads()
```

Computing unresolved vertices. The candidate set is built based on the nearby cells of the cell containing the query object. Since we have only sent the most promising cells to the GPU, there may exist some data objects outside of the candidate set but are actually the result objects. There also may exist shorter paths through vertices outside of the computed cells. Thus, in addition to the candidate set, we use the GPU to compute a set of unresolved vertices.

Definition 3 (unresolved vertex): Assume a query q and a set S of cells. Let the k^{th} smallest distance of objects in S from q be l. An unresolved vertex v satisfies that (1) $\texttt{dist}(q,v) < l$; and (2) v is on the edge of S. The unresolved range of v is the set of locations in G satisfying $\texttt{dist}(v,\cdot) < l - \texttt{dist}(q,v)$.

Here, a vertex v is said to be on the edge of a set S of cells if there exists an edge of which the source vertex is v and the destination vertex is outside of S. We represent the construction of the unresolved vertices set as the function GPU_Unresolved in Algorithm 4 (Line 9). We will compute the final result from the candidate set and objects within unresolved ranges of the unresolved vertices set.

Algorithm 6: Refine_kNN

```
input : a candidate set $\mathcal{R}_{can}$, unresolved vertices set $\mathcal{U}$
output: kNN result $\mathcal{R}$
1  $l \leftarrow$ k_length($\mathcal{R}_{can}$)
2  for each $v$ in $\mathcal{U}$ parallel do
3      $\mathcal{R}_v \leftarrow$ dijkstra_search($l - Dist(q,v)$)
4  $\mathcal{R}_q \leftarrow$ first_k($k, \bigcup_{v \in \mathcal{U}} \mathcal{R}_v \cup \mathcal{R}_{can}$);
```

C. Final Result Refinement

We copy the candidate set and unresolved set to the CPU and refine them to get the final result. This is done by calling the function Refine_kNN in Algorithm 4 (Line 10). Specifically, we perform Dijkstra's algorithm on each unresolved vertex to find the data objects within their unresolved range. Since the Dijkstra's algorithm can be performed for each vertex independently, we use different threads in the CPU to run the algorithm in parallel. Then, we combine the data objects in unresolved range with the candidate set. We return the k data objects with the minimum distances from q as the result. This procedure is summarized in Algorithm 6.

VI. Cost Analysis

A. Space Cost

Given a road network graph G, we store each vertex and edge as a data entry in the graph grid. Although there may be empty entries in the vertex/edge arrays since these arrays have a fixed length, the ratio of empty entries is very low. This is because we map the vertices evenly among cells. Further, we create "virtual vertices" for those vertices with too many edges. The overall space cost of graph grid is $O(|V| + |E|)$.

We can safely delete the messages before $t_{now} - t_{\triangle}$. Assuming that on average, each object sends $f_{\triangle}$ number of location updates during $t_{\triangle}$, the space cost of message lists is $O(f_{\triangle}|\mathcal{O}|)$. In the object table, we store each object as a data entry. Thus, the space cost of the object table is $O(|\mathcal{O}|)$.

B. Query Cost

1) Message Cleaning: The cost of message cleaning consists of two parts: (1) the cost of transferring data from the CPU to the GPU; and (2) the cost of message cleaning in the GPU. These costs are determined by the number of messages to be processed. Given a kNN query and a parameter ρ, the message cleaning aims to compute a candidate set of ρk data objects. We can stop sending more messages to the GPU once the number of candidate objects reaches ρk. Assuming that the average number of messages sent by each object during time $t_{\triangle}$ is $f_{\triangle}$, the number of messages transferred to the GPU is bounded by $O(f_{\triangle}\rho k)$.

Message processing in GPU is performed in parallel. We assign each thread a size-δ^b bucket of messages. To avoid the synchronization issue, we further group every 2^η threads into a bundle. Each bundle works independently from the others and runs `GPU_X_Shuffle` on the threads within it for 2^η iterations. During each iteration, each thread processes $\eta + 1$ messages and updates the intermediate table $\mathcal{T}$ at most $\mu(\eta)$ times. Since η and $\mu(\eta)$ are small constants, the overall cost for message cleaning is $O(\delta^b)$. To compute the final result $\mathcal{R}$ from $\mathcal{T}$, we assign each object in $\mathcal{T}$ with a thread, which processes the candidate messages of the object in $\mathcal{T}$. Since we have $\frac{f_{\triangle}\rho k}{\delta^b 2^\eta}$ candidate messages for each object, the query cost is $O(\frac{1}{\eta}(\log f_{\triangle}\rho k - \log \delta^b))$. As a result, the computation cost of message processing in GPU is $\mathcal{O}(\delta^{\mathrm{b}} + \frac{1}{\eta}(\log f_{\triangle}\rho k - \log \delta^{\mathrm{b}}))$.

Since data transfer and message processing are performed simultaneously, the overall message cleaning cost is $O(f_{\triangle}\rho k)$.

2) Computation of Query Result: To compute the candidate set, the GPU assigns each vertex in C with a thread. Here, C represents the set of cells sent for candidate computation and $|C|$ is proportional to $\rho k/\delta^c$. To run `GPU_SDist`, each thread performs $O(|C|\delta^c\delta^v)$ iterations to update the shortest path distance of the assigned vertex. To find the k nearest objects, the algorithm uses a parallel sorting algorithm that runs in $O(\log \rho k)$ time. To find the set of unresolved vertices, each thread performs a boolean check that runs in $O(1)$ time. Thus, the overall time for computing the candidate set is $O(\rho k \delta^v)$.

In the refinement step, each unresolved range is searched using Dijkstra's algorithm. Assume the objects are uniformly distributed on the road network. Since we run the searching in unresolved ranges in parallel, we only need to consider the time cost of searching the unresolved range of a single vertex. The average search radius is $O(m\sqrt{\frac{k}{\pi}} - \frac{\sqrt{\rho k}}{2})$, where m is the expected ratio of the longest distance to the shortest distance between a query and its k^{th} nearest neighbor. Therefore, the computational complexity of the `dijkstra_search` procedure is $O((m - \sqrt{\rho})\sqrt{k}\log((m - \sqrt{\rho})\sqrt{k}))$.

TABLE II: Statistics of road networks

Dataset	Region	$\|V\|$	$\|E\|$
USA	Full USA	23,974,347	58,333,344
LKS	Great Lakes	2,758,119	6,885,658
CAL	California and Nevada	1,890,815	4,657,742
FLA	Florida	1,070,376	2,712,798
COL	Colorado	435,666	1,057,066
NY	New York City	264,346	733,846

VII. Experimental Study

We evaluate the efficiency of the proposed algorithms empirically in this section.

A. Settings

We implement all algorithms using C++ and CUDA 9.0. We conduct the experiments using a computer with 64-bit Windows operating system, 16GB memory, and two 2.5 GHz Intel Xeon E5 CPUs (12 physical cores in total). We use an Nvidia Quadro P2000 GPU with 1024 cores and 5GB memory.

We used six real-world road networks[1] which are commonly used in related studies [8], [9]. Statistics of the datasets are summarized in Table II. Following a previous study [7], we generate the moving objects using MOTO [10], which is an open-sourced generator for moving object traces. To generate the queries, we randomly generate the query locations and assume a fixed time interval between the queries.

We vary the query parameter k, the number of data objects $|\mathcal{O}|$, and the update frequency of the data objects f. By default, we set $k = 16$, $|\mathcal{O}| = 10^4$, and $f = 1$ (update per second).

We report the average running time of kNN queries. The running time is an amortized time $(T_u + T_q)/n_q$, where T_u is the time for index updating, T_q is the time for query processing, and n_q is the number of queries.

B. Baseline Algorithms

We compare our G-Grid-based algorithm, denoted as **G-Grid**, with two state-of-the-art methods, **V-Tree** [4] and **ROAD** [9]. We extend ROAD to support moving objects following the V-tree paper. In addition, we implement a GPU based version of the V-Tree algorithm, denoted as **V-Tree (G)**.

[1]http://www.dis.uniroma1.it/challenge9/download.shtml

Specifically, we store the core index structure of V-Tree in the GPU memory. Upon receiving a message, we send it to the GPU immediately. We cache the messages in the GPU until the number of cached messages reaches 32, i.e., the size of a GPU warp. Then, we process the cached messages in parallel.

C. Results

1) Tuning System Parameters: We first find the optimal parameter values for G-Grid.

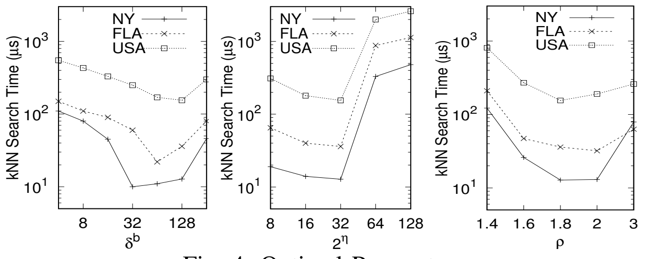


Fig. 4: Optimal Parameters

Optimizing δ^c and δ^v. In the road network datasets, the ratios of the number of edges over the number of vertices are all below 3. Thus, we set $\delta^v = 2$. We use integer (4 bytes) variables for the vertex and edge components. An edge costs 12 bytes, and a vertex costs 32 bytes. To fit a cell in the L1 cache of the CPU (128 bytes), we set δ^c as 3. Thus, a cell costs $32 \times 3 + 8 = 104$ bytes. We pad each cell with 24 extra bytes to ensure its starting address to be a multiple of 128.

If a vertex has more than δ^v edges, we first store the edges into the 24 extra bytes of its cell. When the extra bytes are filled, we create virtual vertices as described in Section III-A.

Optimizing δ^b. We empirically find the optimal value of δ^b. We vary the value of δ^b from 4 to 256 on the datasets of NY, FLA, and US. Fig. 4a shows the result. At start, the running time drops as δ^b increases. This is because a larger δ^b leads to a smaller number of threads and bundles. The size of the intermediate results decreases, which accelerates the `GPU_Collect` process. However, when δ^b becomes too large, the running time increases with δ^b. This is because the number of threads becomes too small to make full use of the parallel capability of the GPU. Based on this set of experiments, we use $\delta^b = 128$ by default in the following studies.

Optimizing 2^η. Parameter 2^η determines the number of threads in each bundle. Intuitively, the more threads in a bundle, the more we can benefit from the `GPU_X_Shuffle` process. However, when the bundle sizes exceeds the GPU warp size, a bundle will contain threads from different warps. Such threads need to be synchronized by calling an expensive function `sync_threads`, the cost of which may outweigh the benefits. As Fig. 4b shows, having more than 32 threads in a bundle leads to worse query performance (32 is the warp size). Thus, we use $2^\eta = 32$ by default.

Optimizing ρ. Parameter ρ balances the workloads of the CPU and the GPU. The optimal value of ρ differs in different hardware platforms. Fig. 4c shows the query running time as we vary the value of ρ from 1.4 to 3. We find that $\rho = 1.8$ suits our hardware settings the best.

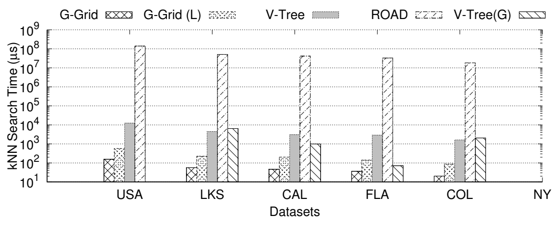


Fig. 5: Query running time vs. datasets

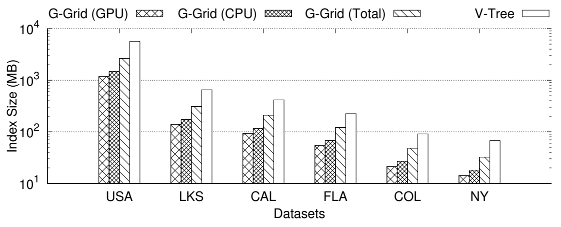
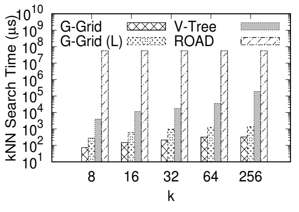


Fig. 6: Index size vs. datasets

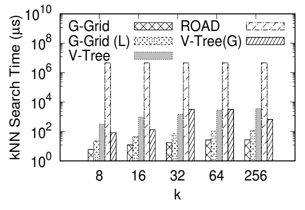
2) Comparing with Baseline Algorithms: Next, we compare the query performance of the different algorithms.

Varying datasets. We compare G-Grid with the baseline algorithms on the six datasets where k is fixed at 16. Figure 5 shows the query running time. For the G-Grid-based algorithm, we report two different times: the average response time of a query, denoted as G-Grid (L), and the amortized time of a query, denoted as G-Grid. Specifically, G-Grid (L) sums up the processing time of every query and reports the average. G-Grid reports the overall response time for processing all the queries divided by the number of queries. Our system can process multiple queries in parallel. Thus, the overall response time is smaller than summing up the response time for each query, i.e., G-Grid is smaller than G-Grid (L). Meanwhile, both G-Grid and G-Grid (L) outperform all the baseline algorithms by orders of magnitude. We omit the result of V-Tree (G) on the USA dataset since its space cost is beyond the capacity of our GPU. We further compare the index sizes of G-Grid with those of V-Tree in Fig. 6. In the figure, G-Grid (CPU) shows the full size of our G-Grid index including the graph grid, the object table, and the message lists; G-Grid (GPU) shows the size of a copy of the graph grid stored in the GPU to streamline the computation; and G-Grid (Total) sums up G-Grid (CPU) and Grid (GPU). The sizes of our full G-Grid indices are significantly smaller than those of the indexes of the more time efficient baseline V-tree. This is because the main component of our index, the graph grid, only stores the original data in a way to facilitate efficient query processing, while the V-tree stores a large amount of precomputed data such as pairwise distances between vertices in a V-tree cell (a subgraph of the network graph [4]) for query processing.

Varying k. We vary k from 8 to 256 and show the results on the USA and NY datasets. We report the running time of each algorithm in Fig. 7. It shows that G-Grid again outperforms the baseline algorithms consistently. The running time of G-Grid and V-Tree increases with k, since the search range in the two algorithms increase with k. When $k > 64$, V-tree (G) becomes more efficient than V-Tree on the NY dataset since it can distribute workloads among parallel threads. ROAD is most costly but its performance is less impacted when k changes. This is because query processing only contributes a small proportion of the running time of ROAD. Most of the workloads in ROAD come from frequent updates of the index structure. We omit the results on the other datasets for conciseness as they show similar patterns.



(a) USA Dataset

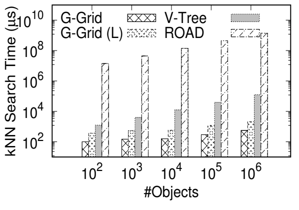


(b) NY Dataset

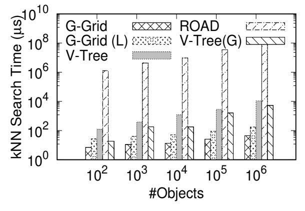
Fig. 7: Varying k

Varying $|\mathcal{O}|$. We vary $|\mathcal{O}|$ from 10^2 to 10^6. As shown in Fig. 8, the four running time of the four algorithms increase as the number of data objects increases. G-Grid increases by a factor of less than 10, which is slower than the baseline algorithms which increase by a factor of around 100.

Varying f. We vary the average object location update frequency f. As shown in Fig. 9, the running time of G-Grid are less impacted by f. In contract, the running time of the baseline algorithms increase rapidly with the update frequency. This confirms the effectiveness of our proposed "lazy update" strategy. By caching the object updates, we successfully reduce the update costs and increase the query efficiency.

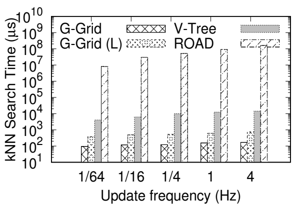


(a) USA Dataset

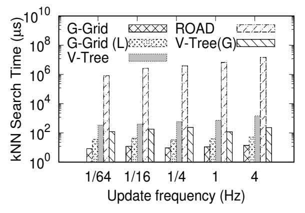


(b) NY Dataset

Fig. 8: Varying $|\mathcal{O}|$



(a) USA Dataset

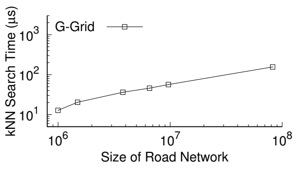


(b) NY Dataset

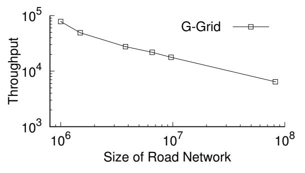
Fig. 9: Varying f

3) Scalability of G-Grid: We further study the scalability of G-Grid over road networks with different sizes.

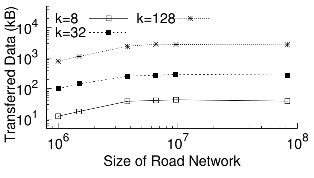
Running time and throughput. We compare the running time and the throughput of G-Grid on the six real road networks with the same number of randomly distributed moving objects. From Fig. 10 (a) and (b), we can see that the running time increases with the size of the road networks. This is because as the network size increases, the objects have larger distances from each other, which leads to a larger search space for distance computations and nearest neighbor finding. We also record the throughput, i.e., number of queries processed by G-Grid per second. As expected, the throughput decreases as the size of road network increases.



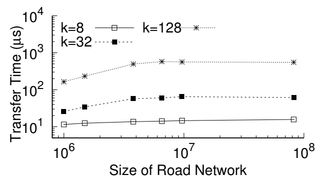
(a) Running time



(b) Throughput



(c) CPU-GPU transfer data size



(d) CPU-GPU transfer time

Fig. 10: Scalability of G-Grid

DRAM-GPU transfer costs. Figure 10 shows the size of the data transferred between the CPU and the GPU and the time for the data transfer, for a query where $k = 8$, 32, and 128. The size of the data transferred increases with k, since the number of message lists and kNN candidates to be sent to the GPU increases with k. The size of the data transferred also increases with the network size. This is because, on a larger road network, the algorithm needs to check more cells to find the kNN candidates. However, the size of the data transferred becomes relatively stable when the networks are sufficiently large (e.g., larger than 10^7), as there are more cells found to be with empty message lists. As for the data transfer time, it has a similar change pattern to that of the size of the data transferred, since the data transfer time is proportional to the size of the data transferred in most cases.

VIII. Related Work

KNN queries have been studied in a variety of contexts [11], [12]. We review two most relevant groups of kNN studies: kNN queries in road networks and kNN queries using GPUs. We also review studies on moving object indexing.

K*NN queries in road networks.* Kolahdouzan and Shahabi [13] precompute *Voronoi cells* over a road network. Then, a kNN query can be processed by first identifying the Voronoi cell within which the query object locates and then expanding the search towards the neighboring Voronoi cells. Huang et al. [14] propose a model for the abstract functionality of a road network NN search. They propose an algorithm similar to Dijkstra's algorithm to compute the distance between the data objects to help identify the NNs online. Papadias et al. [15] utilize the Euclidean distance to facilitate NN search in road networks. A few other studies [16], [17], [18] consider monitoring a kNN query continuously as the objects are moving. Their focus is to reduce the costs of handling the updates relevant to the same query rather than reducing the update costs for all the data objects as in our study.

K*NN queries using GPUs.* GPUs have been used to accelerate *k*NN query processing [19], [20]. A fine-tuned linear algebra library *CUBLAS* [21] is utilized in these studies to compute the distances between objects. Another GPU-based *k*NN algorithm [22] speeds up *k*NN computation for a set of queries using shared computation. This algorithm does not focus on handling object location updates.

Indexing and querying moving objects. More recently, the *cover field tree* [23] is proposed to speed up moving object indexing by controlling the size of each cells in the index. Sidlauskas et al. [24] point out that for update-intensive workloads such as indexing moving objects, grid-based structures outperforms tree-based structures. Darius et al. [7] propose the *PGrid*, which is a main memory index that exploits parallelism of modern multi-core processors to support both long-running queries and rapid updates in Euclidean space. Their work differs from ours in three aspects. First, they focus on *k*NN queries in Euclidean space rather than road networks. Second, they improve the update efficiency through accelerating every update but not delaying or skipping any updates. Third, they achieve parallel processing using a multi-core CPU while we use the GPU, which allows much more threads but also brings challenges in concurrency control. Ward et al. [25] use GPUs for indexing moving objects but consider a moving join query instead of the *k*NN query. Shen et al. [4] study *k*NN queries over moving data objects in road networks. They propose the *V-Tree* that represents a road network as a tree structure. Each leaf node in the tree represents a subgraph of the road network. They precompute the inner- and inter-node distances to accelerate query processing. They update the V-Tree index whenever an object location changes. As a result, their algorithm may repeatedly update the same index entry. Shang et al. [26] study the *best point detour* query in road networks that finds the *point detour* with the minimum detour cost. Here, a point detour is a temporary deviation from a user's current path which allows the user to visit an additional data point.

IX. Conclusion

We studied the *k*NN query in road networks with data location updates. We presented an indexing method that avoids unnecessary index updates via a lazy update strategy. In particular, we proposed a highly parallel algorithm for efficient handling of cached updates, which enables the lazy update strategy. We further proposed a GPU-CPU collaborating algorithm for *k*NN query processing based on our indexing method. The experimental results show that the proposed algorithm outperforms the state-of-the-art algorithms in both query time and index size.

Acknowledgements

This work is supported by the National Key Research and Development Program of China (2016YFC0801607), the National Natural Science Foundation of China (61433008, 61472071, 61472072), and Australian Research Council (ARC) Discovery Project DP180103332. Yu Gu is the corresponding author of this work.

References

[1] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.
[2] S. Shang, B. Yuan, K. Deng, K. Xie, K. Zheng, and X. Zhou, "Pnn query processing on compressed trajectories," *GeoInformatica*, vol. 16, no. 3, pp. 467–496, 2012.
[3] S. Shang, S. Zhu, D. Guo, and M. Lu, "Discovery of probabilistic nearest neighbors in traffic-aware spatial networks," *World Wide Web*, vol. 20, no. 5, pp. 1135–1151, 2017.
[4] B. Shen, Y. Zhao, G. Li, W. Zheng, Y. Qin, B. Yuan, and Y. Rao, "V-tree: Efficient knn search on moving objects with road-network constraints," in *ICDE*, 2017, pp. 609–620.
[5] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[6] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," *New York: International Business Machines Company New York*, 1966.
[7] D. Šidlauskas, S. Šaltenis, and C. S. Jensen, "Parallel main-memory indexing for moving-object query and update workloads," in *SIGMOD*, 2012, pp. 37–48.
[8] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong, "G-tree: An efficient and scalable index for spatial search on road networks," *TKDE*, vol. 27, no. 8, pp. 2175–2189, 2015.
[9] K. C. Lee, W.-C. Lee, and B. Zheng, "Fast object search on road networks," in *EDBT*, 2009, pp. 1018–1029.
[10] J. Dittrich, L. Blunschi, and M. A. V. Salles, "Indexing moving objects using short-lived throwaway indexes." in *SSTD*, vol. 9, 2009, pp. 189–207.
[11] S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis, "Trajectory similarity join in spatial networks," *PVLDB*, vol. 10, no. 11, pp. 1178–1189, 2017.
[12] S. Shang, L. Chen, Z. Wei, C. S. Jensen, J. Wen, and P. Kalnis, "Collective travel planning in spatial networks," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 5, pp. 1132–1146, 2016.
[13] M. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *VLDB*, 2004, pp. 840–851.
[14] X. Huang, C. S. Jensen, and S. Šaltenis, "The islands approach to nearest neighbor querying in spatial networks," in *SSTD*, 2005, pp. 73–90.
[15] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.
[16] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and W. Yi, "Processing moving knn queries using influential neighbor sets," *PVLDB*, vol. 8, no. 2, pp. 113–124, 2014.
[17] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and Q. Deng, "INSQ: an influential neighbor set based moving knn query processing system," in *ICDE*, 2016, pp. 1338–1341.
[18] Y. Wang, R. Zhang, C. Xu, J. Qi, Y. Gu, and G. Yu, "Continuous visible k nearest neighbor query on moving objects," *Inf. Syst.*, vol. 44, pp. 1–21, 2014.
[19] R. J. Barrientos, J. I. Gómez, C. Tenllado, M. P. Matias, and M. Marin, "knn query processing in metric spaces using gpus," in *ECPP*, 2011, pp. 380–392.
[20] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching," in *ICIP*, 2010, pp. 3757–3760.
[21] C. Nvidia, "Cublas library," *NVIDIA Corporation*, vol. 15, no. 27, p. 31, 2008.
[22] G. Chen, Y. Ding, and X. Shen, "Sweet knn: An efficient knn on gpu through reconciliation between redundancy removal and regularity," in *ICDE*, 2017, pp. 621–632.
[23] H. Samet, J. Sankaranarayanan, and M. Auerbach, "Indexing methods for moving object databases: games and other applications," in *SIGMOD*, 2013, pp. 169–180.
[24] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys, "Trees or grids?: indexing moving objects in main memory," in *SIGSPATIAL*, 2009, pp. 236–245.
[25] P. G. D. Ward, Z. He, R. Zhang, and J. Qi, "Real-time continuous intersection joins over large sets of moving objects using graphic processing units," *VLDB J.*, vol. 23, no. 6, pp. 965–985, 2014.
[26] S. Shang, K. Deng, and K. Xie, "Best point detour query in road networks," in *SIGSPATIAL*, 2010, pp. 71–80.