Towards an Efficient Top-K Trajectory Similarity Query Processing Algorithm for Big Trajectory Data on GPGPUs

Eleazar Leal, Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73019, USA
Email: {eleal,ggruenwald}@ou.edu

Jianting Zhang, Simin You
Dept. of Computer Science
City College of New York
New York City, NY 10016, USA
Email: jzhang@cs.ccny.cuny.edu, syou@gc.cuny.edu

***Abstract*—Through the use of location-sensing devices, it has been possible to collect very large datasets of trajectories. These datasets make it possible to issue spatio-temporal queries with which users can gather information about the characteristics of the movements of objects, derive patterns from that information, and understand the objects themselves. Among such spatio-temporal queries that can be issued is the top-K trajectory similarity query. This query finds many applications, such as bird migration analysis in ecology and trajectory sharing in social networks. However, the large size of the trajectory query sets and databases poses significant computational challenges. In this work, we propose a parallel GPGPU algorithm Top-KaBT that is specifically designed to reduce the size of the candidate set generated while processing these queries, and in doing so strives to address these computational challenges. The experiments show that the state of the art top-K trajectory similarity query processing algorithm on GPGPUs, TKSimGPU, achieves a 6.44X speedup in query processing time when combined with our algorithm and a 13X speedup over a GPGPU algorithm that uses exhaustive search.**

***Keywords*-Trajectory; Trajectory similarity; GPGPU; High performance**

I. Introduction

A trajectory is a polygonal line consisting of the points that a moving object occupies in space as time goes by. One way of constructing these polygonal lines is by periodically sampling the positions of the objects being tracked through the use of location sensors like GPS. Given the ubiquity of location sensors, it has been possible to collect large trajectory datasets like the Microsoft Geolife dataset [1], which consist of over 17,621 trajectories and over 23,667,828 points. These datasets make it possible to obtain information about the past, present and future states of the movements of the objects being tracked, and to extract patterns from these movements. Among the different types of queries that can be posed against these trajectory datasets is the top-K trajectory similarity query. A top-K trajectory similarity query receives as inputs a positive integer $K > 0$, and two sets of trajectories, P (the query set) and Q (the database), and returns for every $p \in P$ a set of K trajectories in Q that are most similar to p. An example of a top-K trajectory similarity query is "given the trajectories of my last world trips, find 10 friends with the most similar travel trajectories."

Top-K trajectory similarity queries have many applications. For example, in ecology, scientists are interested in understanding how diseases are transmitted between birds, and how bird species make use of space [2]. These queries also have applications in online social networking sites [1] that allow sharing of travel trajectories. For example, an individual might want to meet other people with the most similar travel trajectories to his own trajectories. These applications involve big trajectory data where data are long trajectories with many locations, and the number of trajectories is large due to the high number of moving objects. In this paper, we refer to these applications as Big Trajectory Data applications.

Processing top-K trajectory similarity queries poses significant computational challenges stemming from three factors: the massive size of the datasets that are of interest for the applications, the internal complexity of a trajectory, and the computational complexity of the similarity measure. One strategy that can be used to tackle these computational challenges consists in exploiting parallel computer architectures, such as GPGPUs.

Among the many advantages of GPGPUs are that they are present in many kinds of computers, from mobile devices to supercomputers; on certain algorithms that exhibit lots of parallelism they can achieve up to an order of magnitude of higher floating point instruction throughput than multicore CPUs [3]; and they are very energy efficient. Another advantage of GPGPUs is that there are works [4] that allow GPGPU processing from within the Spark parallel computing framework [5], so that the high instruction throughput of GPGPUs can be combined with the scalability, ease of use and fault-tolerance of the popular Spark framework. All these advantages of GPGPUs make them adequate tools for tackling the computational challenges associated with processing top-K trajectory similarity queries. GPGPUs have, however, a relatively small memory space, which can be a limitation when processing Big Trajectory Data.

While there are many serial algorithms that deal with

top-K trajectory similarity queries (e.g. [6], [7] [8], and [9]), very little research has been done in this area for GPGPUs. Recently, an algorithm called TKSimGPU [10] was proposed to solve this problem. While TKSimGPU has been shown to work well with small data sets, it does not scale for Big Trajectory Data applications. This is because it generates many spurious candidate trajectory pairs that may not fit into the GPGPU's small memory space.

A key issue when processing top-K trajectory similarity queries on Big Trajectory Data is to avoid unnecessary computations of the similarity measure on trajectory pairs (p, q). This is because most similarity measures have quadratic time complexity on the number of points of p and q, so it is a very expensive operation when the numbers of the trajectories in the query set (P) and in the database (Q) are very large, as it is the case in Big Trajectory Data applications. Additionally, top-K trajectory similarity queries have result sets that have a fixed size $K \times |P| \ll |P \times Q|$, so perfoming an exhaustive search to answer this query requires many unnecessary calculations of the similarity measure on spurious pairs. Therefore, for scalably processing this type of query, it is desirable to reduce the size of the candidate sets involved. *In this paper we introduce Top-KaBT, a GPGPU technique to reduce the number of spurious candidate trajectory pairs generated by Top-K trajectory similarity query algorithms for Big Trajectory Data applications*. This reduction is achieved by calculating the lower and upper bounds of the Hausdorff distance between p and q for every candidate (p, q) pair, and then using the knowledge of these bounds to remove the candidates that for sure cannot form part of the query result set. Top-KaBT was also designed to exploit GPGPUs by ensuring load balancing across the threads, by ensuring memory coalescing, and by using special pruning techniques that reduce the size of the candidate set, thus helping to improve the time performance of the algorithm. We also present an experimental performance study comparing TKSimGPU when combined with Top-KaBT, to reduce candidate sets, against TKSimGPU and a naïve algorithm on GPGPUs that requires an exhaustive search on the set $P \times Q$ using a real trajectory dataset [1].

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 formally introduces the concepts of trajectory and Hausdorff distance, and top-K trajectory similarity queries. Section 4 contains the description of the Top-KaBT algorithm and its mathematical foundations. Section 5 describes the experiments used to study the performance improvements of Top-KaBT along with their results. Finally, Section 8 presents conclusions and future research directions.

II. Related Work

The works ERP [6], EDR [7], wDF [8] and EDwP [9] propose new trajectory similarity metrics and serial algorithms for trajectory similarity queries. Unlike these previous works, our technique, Top-KaBT, reduces the size of the candidate sets generated by techniques that compute trajectory similarities using the Hausdorff distance, which is a commonly used trajectory similarity measure [11]. This distance has the advantage of being a metric. This property is shared with ERP and wDF, and allows the use of the triangular inequality as a means for reducing the amount of work. On the other hand, EDR and EDwP are not metrics; so they need indexes specifically designed for them. The Hausdorff distance also has the advantage that it can be easily extended to mitigate the impact of noisy measurements [11]. However, the pruning ideas behind Top-KaBT are valid not only for the Hausdorff distance, but also for any trajectory similarity measure that satisfies the triangular inequality, like ERP and wDF. The work UTGrid [12] answers top-K trajectory similarity queries (KSQ) on uncertain trajectories with serial algorithms. However, all these techniques are specifically designed for single-core machines, and not for parallel architectures. In order to obtain a scalable parallel algorithm for big trajectory data applications, the algorithm must be designed to exploit the underlying architecture by ensuring load balancing and adequate memory access patterns. This means that all the algorithms discussed in this section would need substantial modifications to run on parallel architectures.

There are far fewer parallel techniques for trajectory similarity queries. U2STRA [13] uses the Hausdorff distance as a similarity measure for near-join trajectory similarity queries on urban trajectories, while the work in [14] uses the Euclidean distance for the same type of query when applied to the study of moving galaxies. The near-join trajectory similarity query is different from the top-K trajectory similarity query because the former takes as input parameters a trajectory query set P, a database Q, and a real number $\epsilon > 0$, and seeks to obtain for every $p \in P$ the set of all $q \in Q$ whose similarity with p is at least ϵ [15].

Another parallel technique for trajectory similarity queries is TKSimGPU [10], which answers top-K trajectory similarity queries on GPGPUs using the Hausdorff distance. A disadvantage of TKSimGPU is that, despite the fact that it exploits the massive parallelism of GPGPUs and avoids an exhaustive search, it may still potentially generate a large number of spurious candidates (as large as $P \times Q$). Such a set of candidates could be too big to fit in the GPGPU's memory, making this technique not suitable for Big Trajectory Data applications. Our proposed technique, Top-KaBT, addresses this issue with its pruning strategies that specifically reduce the size of the candidate sets.

III. Background

In this section we present the definitions of a trajectory, Hausdorff distance, and top-K trajectory similarity queries.

A. Definition of a Trajectory

Given a set $\{(x_i, y_i, t_i) \mid t_i \leq t_{i+1}, 1 \leq i < n\}$ of points in $\mathbb{R}^3$ sampled from the movement of an object with a location sensor, a trajectory over S is a continuous function $\tau : [1, n] \rightarrow \mathbb{R}^3$ where $\tau(i) = (x_i, y_i, t_i)$ for all integers $i \in [1, \ldots, n]$ and such that $\tau(x)$, with $x \in [t_i, t_{i+1})$, is the interpolated value between $\tau(i)$ and $\tau(i+1)$ [16].

B. Definition of Hausdorff Distance

Given two finite sets of points P and Q, the Hausdorff distance $hausd(P, Q)$ between P and Q is defined as the maximum between $\max_{p \in P} \min_{q \in Q} d(p, q)$ and $\max_{q \in Q} \min_{p \in P} d(p, q)$ [11].

C. Definition of Top-K Trajectory Similarity Query

Given a positive integer $K > 0$, two finite non-empty sets of trajectories P (the query set) and Q (the database), and a similarity measure $\sigma : S \times S \rightarrow R$, a top-K trajectory similarity query returns for every $p \in P$ a set R_p satisfying that $|R_p| = K$ and for every $q \in R_p$ and $q_{other} \in Q - R_p$ it is the case that $\sigma(q_{other}, p) \leq \sigma(q, p)$ [8].

IV. The Top-KaBT Algorithm

A. Overview

Top-KaBT is a parallel GPGPU algorithm for reducing the number of spurious candidate trajectory pairs (p, q) generated by top-K trajectory similarity query GPGPU algorithms that follow the filter-and-refine schema and use a trajectory similarity that satisfies the properties of a metric. To accomplish this, Top-KaBT calculates lower and upper bounds of the Hausdorff distance between p and q for every candidate pair (p, q). These calculations are much cheaper than the calculations of the Hausdorff distances, a fact which will be proved in Section IV.B. After this, Top-KaBT sorts the pairs according to their lower bounds of the Hausdorff distance, and uses these bounds to remove spurious candidate pairs. By removing spurious candidate pairs, this technique lessens the negative impact of the small size of the GPGPU's memory, and reduces the time wasted computing the similarity for these spurious pairs. Additionally, the technique addresses load balancing and memory coalescing, by having threads within a thread block perform the same amount of work, and by having threads with consecutive indices access adjacent memory locations.

B. Theoretical Foundations of Top-KaBT's Pruning Strategy

In this section we present the definitions and theorems on which this pruning technique rests. The main result is Theorem 7, which states that if we have a trajectory p with n_p candidate pairs $C_p = \{(p, q_0), (p, q_1) \ldots, (p, q_{n_p - 1})\}$ sorted by the lower bounds to their respective Hausdorff distances, then if we find an integer $0 \leq v_K < n_p$ meeting certain conditions explained later, we'll know that the K most similar trajectories to p will be among $\{(p, q_0), (p, q_1), \ldots, (p, q_{v_K})\}$, and we can prune the remaining elements $\{(p, q_{v_K+1}), (p, q_{v_K+2}), \ldots, (p, q_{n_p-1})\}$.

Notation: In the remainder of the paper, we denote by P the set of query trajectories, Q the trajectory database, and use m_{p,q_i} to refer to $\min_{x \in MBR(p), y \in MBR(q_i)} d(x, y)$, where $(p, q_i) \in P \times Q$ and $d(x, y)$ is the Euclidean distance between points x and y. Similarly, we use the notation M_{p,q_i} to refer to $\max_{x \in MBR(p), y \in MBR(q_i)} d(x, y)$.

Lemma 1. *For any $(p, q) \in P \times Q$, it is the case that $m_{p,q} \leq hausd(p, q) \leq M_{p,q}$.*

Proof: Let a and b be points such that $a \in MBR(P)$, $b \in MBR(Q)$, and $hausd(p, q) = d(a, b)$. By definition of $m_{p,q}$ we have that $m_{p,q} = min_{x \in MBR(p), y \in MBR(q)} d(x, y) \leq d(a, b) = hausd(p, q)$. The proof of $hausd(p, q) \leq M_{p,q}$ is analogous. ■

Definition 2 (Cut point set)**.** *Given the candidate set $C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p-1})\}$, with $m_{p,q_i} \leq m_{p,q_{(i+1)}}$ for $0 \leq i < n_p - 1$, the cut-point set of C_p is defined as $CP_p = \{i \in \mathbb{Z} \mid M_{p,q_i} \leq m_{p,q_{(i+1)}}\}$. The elements of the cut-point set are called* cut-points*.*

Example 3. *If $C_p = \{(p, q_0), (p, q_1), (p, q_2), (p, q_3)\}$ such that $m_{p,q_0} = 2.2$, $m_{p,q_1} = 2.3$, $m_{p,q_2} = 3.3$, $m_{p,q_3} = 4.1$, and $M_{p,q_0} = 2.4$, $M_{p,q_1} = 2.7$, $M_{p,q_2} = 4.0$, $M_{p,q_3} = 4.2$, then $CP_p = \{1, 2\}$ because $M_{p,q_1} = 2.7 \leq 3.3 = m_{p,q_2}$, and $M_{p,q_2} = 4.0 \leq 4.1 = m_{p,q_3}$.*

Definition 4. *Given the candidate set $C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p-1})\}$, with $m_{p,q_i} \leq m_{p,q_{i+1}}$ for $0 \leq i < n_p - 1$, with cut-point set $CP_p \neq \emptyset$, the min-cut point of C_p is defined to be* $\min CP_p$.

Definition 5 (min-K-Cut point)**.** *Given the candidate set $C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p-1})\}$, with $m_{p,q_i} \leq m_{p,q_{i+1}}$ for $0 \leq i < n_p - 1$, with 1-cut-point set $CP_p \neq \emptyset$, the min-K-cut point of C_p is defined to be the K-th smallest element in CP_p.*

Theorem 6. *If v is a cut point of the candidate set $C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p-1})\}$, with $m_{p,q_i} \leq m_{p,q_{i+1}}$ for $0 \leq i < n_p - 1$, then the 1-nearest neighbor to trajectory p is a q_i with $0 \leq i \leq v$.*

Proof: Assume that v is a cut point of C_p. Then, $M_{p,q_v} \leq m_{p,q_{v+1}}$ is true, and since $m_{p,q_{v+1}}$ is a lower-bound of $hausd(p, q_{v+1})$, and $M_{p,q_{v+1}}$ is an upper bound of $hausd(p, q_{v+1})$, then we have the inequality $hausd(p, q_v) \leq M_{p,q_v} \leq hausd(p, q_{v+1})$. By induction, we can easily prove that $hausd(p, q_v) \leq hausd(p, q_j)$ for $v \leq j < n_p$. Therefore, the 1-nearest neighbor to p must be a q_i with $0 \leq i \leq v$, which is what we wanted to prove. ■

Theorem 7. *If v_K is a min-K-cut point of the candidate set $C_p = \{(p, q_0), (p, q_2), \cdots, (p, q_{n_p-1})\}$, with $m_{p,q_i} \leq m_{p,q_{i+1}}$ for $0 \leq i < n_p - 1$, then the top-K nearest neighbors*

of trajectory p lie among the q_i with $0 \leq i \leq v_K$.

Proof: We proceed by induction on K. The base case with $K = 1$ has already been proved in the previous theorem. Assume $k > 1$ and that the theorem holds for $K = k$. Let's verify that the theorem holds for $K = k + 1$. Let v_k and v_{k+1} be the min-k-cut and the min-$(k+1)$-cut points of C_p, respectively. By inductive hypothesis, we know that the k nearest neighbors of p are contained in the set $\{q_i \mid 1 \leq i \leq v_k\}$. We also know that, by the definition of min-K cut point, $v_k \leq v_{k+1}$, and also that $hausd(p, q_{k+1}) \leq hausd(p, q_j)$ for $k + 1 \leq j < n_p$. This implies that the $k + 1$ nearest neighbors of p are in the set $\{q_i \mid 0 \leq i \leq v_{k+1}\}$, which is what we wanted to prove. ■

Example 8. *Continuing with Example 3 and using Theorem 7, we know that the top-2 nearest neighbors of trajectory p are contained in the set $C_p = \{(p, q_0), (p, q_1), (p, q_2)\}$. The theorem allows us to discard the candidate pair (p, q_3).*

Observation 9. *The minimum Euclidean distance between two MBRs R with lower-left corner (r_x, r_y) and upper left corner (r'_x, r'_y), and S with lower-left corner (s_x, s_y) and upper left corner (s'_x, s'_y), can be computed in constant time complexity using the mindist formula of [17]: $mindist(R, S) = \sqrt{d_x^2 + d_y^2}$, where $d_i = r_i p_i$ if $p_i < r_i$, $d_i = p_i - r'_i$ if $r'_i < p_i$, and $d_i = 0$ otherwise, for $i \in \{x, y\}$. Similarly, the maximum Euclidean distance between R and S can be found using $maxdist(R, S) = \sqrt{c_x^2 + c_y^2}$, where $c_i = r'_i p_i$ if $p_i < (r_i + r'_i)/2$, and $c_i = p_i - r'_i$ otherwise.*

Observation 10. *Given a candidate set $C_p = \{(p, q_0), (p, q_2), \cdots, (p, q_{n_p-1})\}$, Observation 9 can be used to efficiently compute m_{p,q_i} and M_{p,q_i} because these two represent the minimum and maximum Euclidean distances between the MBRs of trajectories p and q_i, for any $(p, q_i) \in C_p$.*

C. Description of Top-KaBT's Pruning Strategy

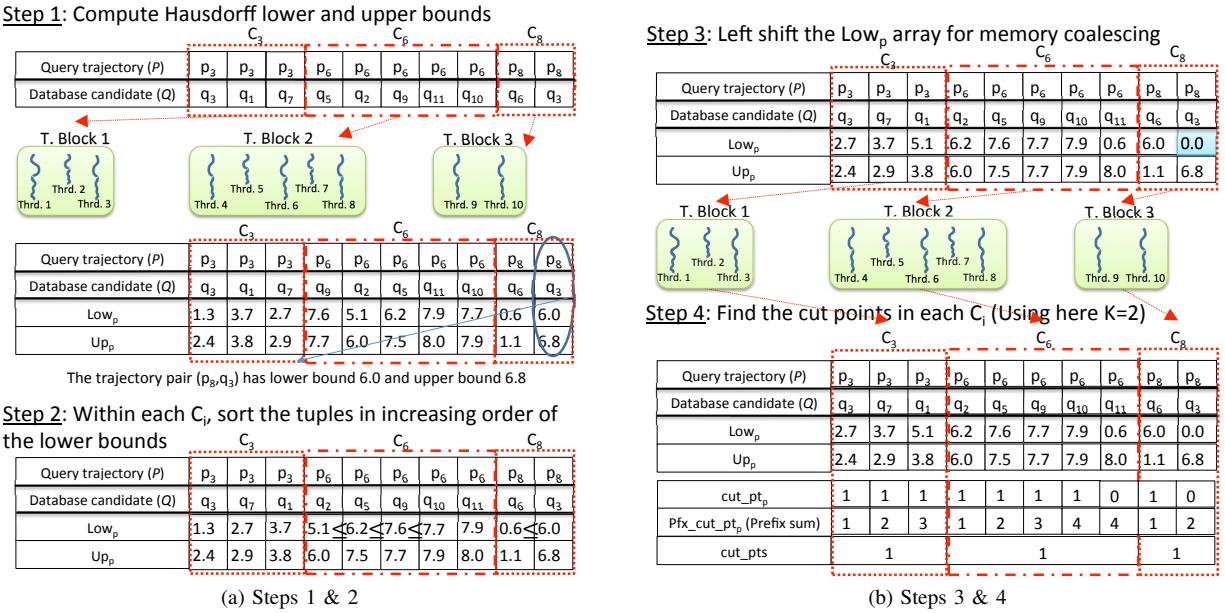
In this subsection we describe our proposed parallel GPGPU technique to prune the candidate set of the top-K trajectory similarity query processing algorithm. The pseudocode algorithm for the pruning technique is in Algorithm 1, while Fig. 1, 2 and 3 provide an illustrated example.

The function SORT_PRUNING in Line 1 of Fig. 1 is in charge of further pruning the set of (p, q) candidate pairs, by removing pairs that cannot form part of the result set, as assured by Theorem 7. This function takes an integer K and a list of (p, q) pairs $candidates$ as input and returns as output a sub-list of $candidates$. In Line 3 we consider Q_p the set of all q trajectories that up to this point have been identified as possible candidates for being the most similar Q-trajectories to p. Then Line 4 calculates the lower and upper bounds (low_p and up_p, respectively) of the trajectory similarity between p and q, using Observation 10. This is illustrated in Fig. 1 Step 1, where we can see that different thread blocks are assigned different p query trajectories, and every thread in a thread block is in charge of a different (p, q) trajectory pair. The first thread block is in charge of finding the lower and upper bounds of the Hausdorff distance for each of the pairs (p_3, q_3), (p_3, q_1) and (p_3, q_7). Line 5 sorts the arrays Q_p, low_p, and up_p, using the entries in low_p as keys; in this way we ensure that the premise of Theorem 7 is satisfied. An example of this is shown in Fig. 1 Step 2, where we see that the pairs corresponding to the first thread block have been sorted according to their lower bounds so

Algorithm 1 Prunes top-K trajectory similarity candidate set

Input: An integer $K > 0$. A set $candidates \subseteq P \times Q$.
Output: A set $Result \subseteq candidates \subseteq P \times Q$ that contains the result of a top-K trajectory similarity query.

1: **function** SORT_PRUNING($K, candidates$)
2: **for** $p \in \Pi_P(candidates)$ **do in parallel**
3: $Q_p \leftarrow \{q \in Q \mid (p, q) \in candidates\}$
4: $(low_p, up_p) \leftarrow hausdorff\_bounds(p, Q_p)$
5: $\widehat{Q}_p, \widehat{low}_p, \widehat{up}_p \leftarrow$ Sort Q_p, low_p, up_p using low_p as key
6: $\widehat{low}_p \leftarrow left\_shift\_array(\widehat{low}_p)$
7: $cut\_pt_p \leftarrow find\_cut\_point(p, K, \widehat{low}_p, \widehat{up}_p)$
8: $\widehat{candidates}_p \leftarrow \{(p, q) \mid q \in \widehat{Q}_p\}$
9: **end for**
10: $\widehat{candidates} \leftarrow \cup_{p \in \Pi_P(candidates)} \widehat{candidates}_p$
11: $cut\_pts \leftarrow \cup_{p \in \Pi_P(candidates)} cut\_pt_p$
12: $Result \leftarrow remove(\widehat{candidates}, cut\_pts)$
13: **return** $Result$
14: **end function**
15: **function** HAUSDORFF_BOUNDS(p, Q_p)
16: $QMBR_p \leftarrow \{MBR(q) \mid q \in Q_p\}$
17: **for** $i \in \{0, 1 \ldots, |Q_p| - 1\}$ **do in parallel**
18: $low_p[i] \leftarrow \min d(MBR(p), QMBR_p[i])$
19: $up_p[i] \leftarrow \max d(MBR(p), QMBR_p[i])$
20: **end for**
21: **return** (low_p, up_p)
22: **end function**
23: **function** FIND_CUT_POINT(p, K, low_p, up_p)
24: **for** $i \in \{0, 1, \ldots, |low_p| - 1\})$ **do in parallel**
25: **if** $up_p[i] \leq low_p[i]$ **then**
26: $cut\_pt_p[i] \leftarrow 1$
27: **else**
28: $cut\_pt_p[i] \leftarrow 0$
29: **end if**
30: **end for**
31: $Pfx\_cut\_pt_p \leftarrow InclPfxSum(cut\_pt_p)$
32: $cut\_pt_p \leftarrow \min\{i \mid Pfx\_cut\_pt_p[i] = K\}$
33: **return** cut_pt_p
34: **end function**
35: **return** $sort\_pruning(K, candidates)$



(a) Steps 1 & 2 (b) Steps 3 & 4

Figure 1. Example run of the SORT_PRUNING step of Top-KaBT (Steps 1 to 4)

Algorithm 1 Prunes top-K trajectory similarity candidate set

36: **function** REMOVE($candidates, cut\_pts$)
37: $n \leftarrow |\Pi_P(candidates)|$
38: **for** $i \in \{0, 1, \ldots, n-1\}$ **do in parallel**
39: $offset[i] \leftarrow |\{(p_j, q) \in candidates \mid p_j < p_i\}|$
40: $B[2i] \leftarrow cut\_pts[i]$
41: $B[2i+1] \leftarrow |\Pi_P(candidates)[i]|$
42: $Alter\_1s0s[2i] \leftarrow 1$
43: $Alter\_1s0s[2i+1] \leftarrow 0$
44: **end for**
45: $Counts \leftarrow adjacentDifference(B)$
46: $Stencil \leftarrow RLD(Counts, Alter\_1s0s)$
47: $PfxStencil \leftarrow ExclPfxSum(Stencil)$
48: **for** $i \in \{0, \ldots, n-1\}$ **do in parallel**
49: **if** $Stencil[i] = 1$ **then**
50: $Pruned[PfxStencil[i]] \leftarrow candidates[i]$
51: **end if**
52: **end for**
53: **return** $pruned$
54: **end function**

that (p_3, q_3) has smaller lower bound (whose value is 1.3) than (p_3, q_7), which has 2.7 as a lower bound, and (p_3, q_7) in turn has a smaller lower bound than (p_3, q_7), which has a lower bound of 3.7. In Line 6 of Fig. 1, low_p is shifted 1 entry to the left for memory coalescing in Line 23. The reason for this is that, according to Theorem 7, we test if $M_{p,q_i} \leq m_{p,q_{i+1}}$ for every p and q_i, so the value $low_p[0]$ corresponding to m_{p,q_0} is never used. Fig. 1 Step 3 shows the left-shifting of Low_p. Notice how the first value (1.3) of the lower bounds array disappeared, and we added a 0.0 to the right of the same array. Because of Theorem 7, this last value we added to the right is never used. Line 7 finds the cut point associated with every p query trajectory using the lower and upper bounds of the trajectory similarity measure. This corresponds to Step 4 in Fig. 1 and Step 5 in Fig. 2.

The function HAUSDORFF_BOUNDS in Line 15, shown in Step 1 of Fig. 1, receives a trajectory p, and a list Q_p with the associated Q trajectory candidates, and finds low_p and up_p that satisfy: $low_p \leq Hausd(p, q) \leq up_p$. In Lines 17 to 20, low_p and up_p are computed in parallel for every $q \in Q_p$ using Observation 10. This function does memory coalescing when writing the bounds of the MBRs back to the global memory because threads with consecutive identifiers write the MBR bounds of trajectories with consecutive indexes. This function also achieves load balancing within thread blocks because the complexity of computing the MBRs does not depend on the trajectories themselves; therefore, all threads perform the same amount of work.

Function FIND_CUT_POINT in Line 23 receives as input parameters a $p \in P$, an integer $K > 0$, and the two arrays low_p and up_p of the lower and upper bounds, respectively, and is in charge of finding the smallest K-cut point using Theorem 7. After the parallel loop in Lines 24 through 30, an array cut_pt_p is obtained, which is shown in Fig. 1 Step 4. There we see that the cut_pt_p boolean array has the value 1 at position i if the corresponding pair has an index that is a cut point, and 0 otherwise. For example, in the pairs associated with the second thread block, the cut_pt_p entry associated with the pair (p_6, q_{11}) is 0 because $0.6 < 8.0$. To find the smallest K-cut point for p, a parallel exclusive prefix sum [18] over cut_pt (which is the portion of the

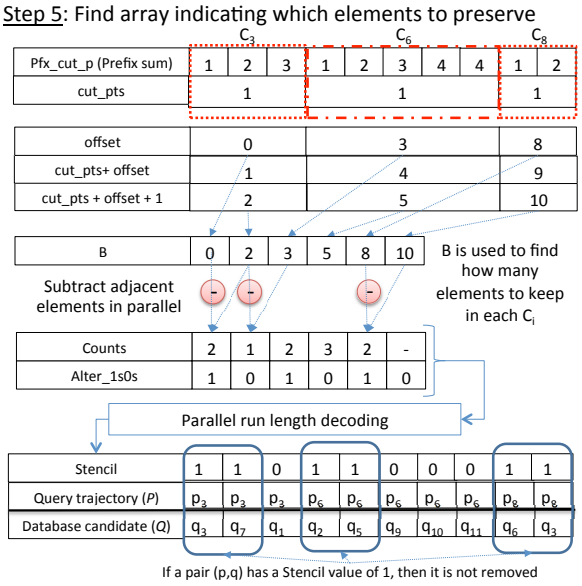


Figure 2. Example run of SORT_PRUNING (Step 5)

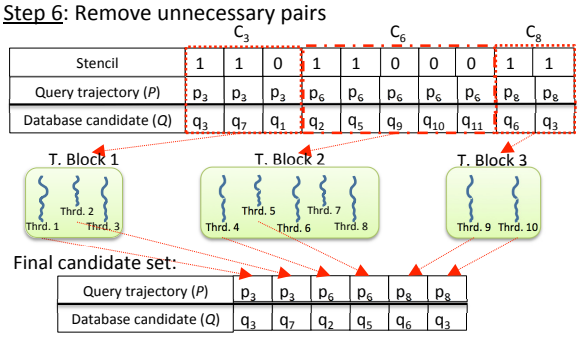


Figure 3. Example run of SORT_PRUNING (Step 6)

cut_pt_p array corresponding to p) is performed to obtain the array Pfx_cut_pt of Line 31; this is shown in Fig. 2 where the second thread block obtained the array $[1, 2, 3, 4, 4]$. After this, every thread block finds the smallest index i such that $Pfx\_cut\_pt_p[i] \geq K$. In the case of the second thread block, the first i that satisfies this condition is $i = 1$ because there is a 2 in the Pfx_cut_pt portion of the second thread block at position 1. This function performs memory coalescing because threads with consecutive indexes access adjacent memory locations in the cut_pt_p array. Also, all threads perform the same amount of work.

Function REMOVE in Line 36 receives as input parameters the array $candidates$ with the candidate trajectory pairs (p, q), and an array cut_pts of length $|\Pi_p(candidates)|$ (where $\Pi_p(candidates)$ is the projection on the left component (P) of the tuples in $candidates$), one for every p candidate. This last array satisfies that $cut\_pts[i]$ is the cut point associated with the i-th p trajectory in $\Pi_p(candidates)$. Lines 38 through 44 create an array B that contains the elements of $cut\_pts+offset+1$ in its even-indexed entries, and the elements $|\{(p_j, q) \in candidates \mid p_j < p_i\}|$ in its odd-indexed entries i. In Fig. 2 we see in Step 5 that the elements of $cut\_pts + offset + 1$ are $B[1] = 2$, $B[3] = 5$ and $B[5] = 10$. The idea behind creating B is to count how many pairs (p, q) need to be preserved for every p. These same Lines 38 to 44, create another array $Alter\_1s0s$ with 1s in its even entries and 0s in its odd entries. This array is used for run-length decoding [19]. Then Line 45 calculates the adjacent difference array $Counts$ satisfying that $Counts[i] = B[2i + 1] - B[2i]$. $Counts[2i + 1]$ is the number of Q candidates associated with p_i that can be pruned away, while $Counts[2i]$ indicates the number of Q candidates associated with p_i that cannot be pruned away. In Fig. 2 we see that in Step 5 $Counts[0] = B[1] - B[0] = 2 - 0 = 2$, and $Counts[1] = B[2] - B[1] = 3 - 2 = 1$. This means that 2 pairs associated with p_3 (which is the 0-th p candidate) cannot be pruned away, but 1 pair can be pruned away. Line 46 performs a run-length decoding over $Counts$ (containing the counts of how many times each element occurs in the result of the run-length decoding) and $Alter\_1s0s$ (containing the elements that will be in the result of the run-length decoding); this is to obtain the array $Stencil$, which has a 1 at position i if and only if $candidates[i]$ cannot be pruned, and a 0 at position i if $candidates[i]$ can be safely pruned according to Theorem 4.6. Lines 48 to 52 prune the spurious candidates by writing into $Pruned$ only those elements of $candidates$ located at positions i such that $Stencil[i] = 1$. In Fig. 3 we see that the candidates (p_3, q_1), (p_6, q_9), (p_6, q_{10}) and (p_6, q_{11}) had associated $Stencil$ values of 0; therefore, they were pruned.

V. EXPERIMENTS AND RESULTS

In this section we describe the dataset, the hardware and software environment, and the experiments used to compare TKSimGPU, when combined with Top-KaBT to reduce candidate sets against TKSimGPU itself and against a naïve exhaustive GPGPU search algorithm. The naïve exhaustive search algorithm finds the Hausdorff distances between all pairs of $(p, q) \in P \times Q$, and then sorts those distances to select the top K most similar trajectories in Q for every $p \in P$.

A. Datasets and experiment setup

For our experiments we use the Microsoft GeoLife trajectory dataset [1] consisting of the trajectories described by 182 individuals carrying smartphones, while going through their daily lives. This dataset consists of 17,621 trajectories whose spatial lengths add up to 1.2 million kilometers and whose total time lengths sum up to 48,203 hours. The total number of points in all the trajectories is 23,667,828 points.

In these experiments we use only those trajectories labeled with the keyword "walk" because the trajectories that meet this condition are usually shorter; hence, their

MBRs contain less dead space. Prior to the execution of our experiments, we segment the trajectories in the original dataset by recursively splitting a trajectory if either the object in question remains stationary for longer than 30 minutes, or if it contains more than 256 points. This is because we use MBRs to perform filtering, so trajectories with many points may have very large MBRs that can potentially intersect with all other MBRs. After preprocessing we end up with 18,000,000 tuples (x, y, t) belonging to 86,448 trajectories that are then stored in the GPGPU's global memory.

We use a workstation running 2 Intel Xeon E5 2610v2 chips, 64 GB of RAM, and an nVidia Quadro K5000 GPU. Our code uses Thrust 1.8 [20], CUDA 7.0, and was compiled with g++ with O2 optimizations.

B. Experiments

We now describe three experiments to evaluate the impacts of the size of the query set, the size of the database and the size of the parameter K on the average query processing time of TKSimGPU combined with Top-KaBT, TKSimGPU, and the naïve exhaustive GPGPU search algorithm, which are denoted as TKSimGPU + Top-KaBT, TKSimGPU and naïveGPU, respectively in Figure 4. In all the experiments, we choose a grid cell size of 128×128, and for TKSimGPU+Top-KaBT and TKSimGPU we take a sample size of 512 elements. We also use 512 GPGPU threads. Our experiments show that these parameters give the best query processing times.

1) Impact of the query set size ($|P|$): In this experiment we use a database size (Q) of 40,000 trajectories (whose points add up to 8.6e6), and $K = 70$. We vary the query set size from 20 to 100 trajectories (up to 17e3 points (x, y, t)).

In Fig. 4a we see that the average query execution times of all three techniques are linear. This is because the average query execution time is dominated by the average number of (p, q) candidates that remain before the refinement stage, and this number of candidate pairs grows, in the case of our three techniques, linearly with the size of the query set. This behavior was expected for the naïve implementation because its final candidate set is $P \times Q$, and if Q is fixed, the cardinality of this candidate set is a linear in $|P|$.

In Fig. 4a we see that if the database size is fixed, and the query set size increases linearly, then the average query execution time in TKSimGPU+Top-KaBT is on average 4.72 times faster than in TKSimGPU because the candidate set size of TKSimGPU+Top-KaBT is on average 8 times smaller than the one that TKSimGPU calculates. TKSimGPU is also 11 times faster than naïveGPU because its candidate set size is 15 times smaller than the naïve's.

2) Impact of the database size ($|Q|$): In this experiment we use a query set size of 60 trajectories, and $K = 70$. The database size varies in the range from 28,000 to 56,000 trajectories (from 5e6 points up to 12e6 points (x, y, t)).

In Fig. 4b we observe that the time complexity of the average query execution time for the three techniques is linear in terms of the database size when the query set size and K are kept constant. The reason for this is that the time complexity is dominated by the average number of candidate pairs remaining after pruning, which is linear in $|Q|$.

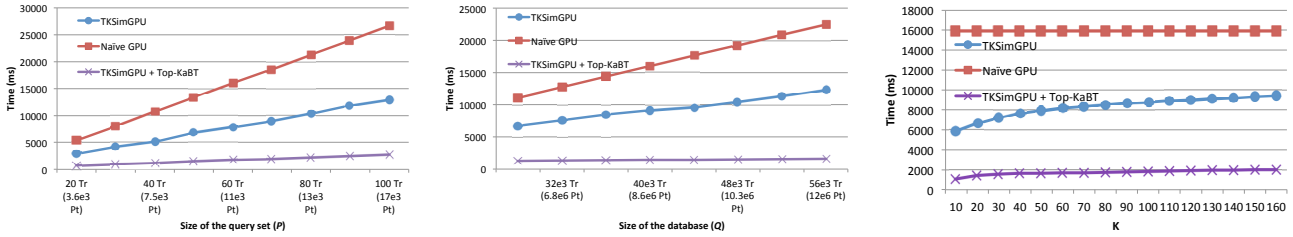
In Fig. 4b we observe that TKSimGPU+Top-KaBT is on average 6.44 times faster than TKSimGPU because the final number of candidate pairs produced by TKSimGPU+Top-KaBT is 11 times smaller than the number of candidate pairs produced by TKSimGPU. The reason why TKSimGPU+Top-KaBT reduces the number of candidate pairs by a constant larger than the constant with which the TKSimGPU+Top-KaBT outperforms TKSimGPU is that TKSimGPU+Top-KaBT pays with the overhead of calculating the K-cut points and compressing the candidate pairs array. In this figure we observe also that TKSimGPU is 13 times faster than naïveGPU because the latter algorithm computes $P \times Q$, while TKSimGPU performs pruning and thus reduces the size of the candidate pairs set.

3) Impact of K: In this experiment we use a query set size of 60 trajectories, a database size of 40,000 trajectories (10,330,000 points (x, y, t)), and vary K from 10 to 160.

In Fig. 4c we observe that the time complexity of the exhaustive search GPU algorithm remains constant, even though it does increase but almost imperceptibly at the scale of the plot, as K increases. The reason for this is that the bulk of the operations of the exhaustive search algorithm consists in calculating $P \times Q$, which is independent of K. Also, the time complexity of TKSimGPU and TKSimGPU+Top-KaBT has a similar shape, where the average query processing time increases quickly for small K, and then the speed of increase stabilizes. Finally, in Fig. 4c we observe that TKSimGPU+Top-KaBT outperforms TKSimGPU in terms of average query processing time, and this latter technique outperforms naïve GPU. This is because, again, the average query processing time is dominated by the size of the candidate pairs set, which exhibits the same behavior observed in Fig. 4c. The size of the candidate pair set of TKSimGPU+Top-KaBT is 7 times smaller than the size of the candidate pair set of TKSimGPU, which in turn is 4 times smaller than that of naïveGPU.

VI. Conclusion

In this paper we proposed Top-KaBT, a parallel technique to reduce the number of spurious candidate trajectory pairs generated when processing top-K trajectory similarity queries for Big Trajectory Data applications on GPGPUs. This reduction is necessary because in Big Trajectory Data applications the number of spurious candidate pairs is typically very large, so it has an associated unnecessary large computational overhead. Top-KaBT works by using only the lower and upper bounds of the similarity measure to remove the candidate pairs that surely cannot belong to the query



(a) Impact of the Query Set Size

(b) Impact of the Database Size

(c) Impact of K

Figure 4. Experiment results

result set. This reduces the negative impact arising from the small size of the GPGPU's global memory. In addition, the technique achieves load balancing and memory coalescing by having threads perform the same amount of work, and by having threads with consecutive indices access consecutive memory locations. We compared the performance of a state of the art top-K trajectory similarity query processing algorithm, TKSimGPU, when combined with our technique to reduce candidate sets against TKSimGPU itself and against naïveGPU, a exhaustive search technique on GPGPU, using a real-life large-scale trajectory dataset. The experiments show that Top-KaBT reduces the size of the candidate set of trajectory pairs by a factor of up to 10X, which leads to a substantial gain in performance in TKSimGPU when combined with Top-KaBT: it achieves a speedup of up to 6.44X in average query processing time compared to TKSimGPU alone and a speedup of 13X compared to naïveGPU.

For future research, we plan to design a parallel technique that uses a trajectory similarity measure that, unlike Hausdorff's, takes the temporal dimension into consideration. This is useful for the travel trajectory sharing applications discussed in Section I because the trajectories of two users could be spatially similar, but very dissimilar in the temporal dimension. For example, if one user usually travels in the spring and the other one in the summer.

Acknowledgment

This work is supported in part by the National Science Foundation under Grant No. 1302439 and 1302423.

References

[1] Y. Zheng, X. Xie, and W.-Y. Ma, "Geolife: A collaborative social networking service among user, location and trajectory," *IEEE Database Engineering Bulletin*, June 2010.

[2] J. S. Horne, E. O. Garton, S. M. Krone, and J. S. Lewis, "Analyzing animal movements using brownian bridges," *Ecology*, vol. 88, no. 9, pp. 2354–2363, 2007.

[3] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," *SIGARCH*, vol. 38, no. 3, pp. 451–460, 2010.

[4] P. Li, Y. Luo, N. Zhang, and Y. Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in *IEEE NAS*, 2015, pp. 347–348.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Hot Cloud*, 2010.

[6] L. Chen and R. Ng, "On the marriage of lp-norms and edit distance," in *VLDB*, 2004.

[7] L. Chen, M. T. Özsu, and V. Oria, "Robust and fast similarity search for moving object trajectories," ser. SIGMOD 2005.

[8] H. Ding, G. Trajcevski, and P. Scheuermann, "Efficient similarity join of large sets of moving object trajectories," in *TIME '08*, 2008, pp. 79–87.

[9] S. Ranu, P. Deepak, A. Telang, P. Deshpande, and S. Raghavan, "Indexing and matching trajectories under inconsistent sampling rates," in *ICDE*, 2015.

[10] E. Leal, L. Gruenwald, J. Zhang, and S. You, "Tksimgpu: A parallel top-k trajectory similarity query processing algorithm for gpgpus," in *IEEE Int'l Conf.on Big Data*, 2015.

[11] S. Nutanong, E. H. Jacox, and H. Samet, "An incremental hausdorff distance calculation algorithm," *Proc. VLDB Endow.*, vol. 4, no. 8, pp. 506–517, 2011.

[12] C. Ma, H. Lu, L. Shou, and G. Chen, "Ksq: Top-k similarity query on uncertain trajectories," *TKDE*, 2013.

[13] 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.

[14] M. Gowanlock and H. Casanova, "Distance threshold similarity searches on spatiotemporal trajectories using gpgpu," in *HiPC*, 2014, pp. 1–10.

[15] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowl. Inf. Syst.*, vol. 7, no. 3, pp. 358–386, Mar. 2005.

[16] H. Cao, O. Wolfson, and G. Trajcevski, "Spatio-temporal data reduction with deterministic error bounds," *The VLDB Journal*, vol. 15, no. 3, pp. 211–228, Sep. 2006.

[17] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," *SIGMOD*, 2000.

[18] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007.

[19] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proc. VLDB*, vol. 3, pp. 670–680, 2010.

[20] J. Hoberock and N. Bell, "Thrust: A parallel template library."