PARALLEL PROCESSING OF TOP-K TRAJECTORY SIMILARITY QUERIES WITH GPGPUS

Eleazar Leal¹, Le Gruenwald¹, Jianting Zhang²

¹School of Computer Science
University of Oklahoma
Norman, OK USA

²Department of Computer Science
City College of New York
New York City, NY USA

eleal@ou.edu; ggruenwald@ou.edu; jzhang@cs.ccny.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. The experiments also show that the time overhead incurred by Top-KaBT is very small because it only represents 2.5% of the average query execution time spent by TKSimGPU when combined with Top-KaBT.

This is an extended work of Leal, Gruenwald, Zhang and You, 2016, which was published in the Proceedings of IEEE International Congress on Big Data (Big Data Congress) 2016.

Keywords: Trajectory; Trajectory similarity; GPGPU; High performance

1. 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 (Zheng, Xie, & Ma, 2010), which consists 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 (Home, Garton, Krone, & Lewis, 2007). These queries also have applications in online social networking sites (Zheng, Xie, & Ma, 2010) 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 to 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; with 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 (Lee et al., 2010); and they are very energy efficient. Another advantage of GPGPUs is that there are works (Li, Luo, Zhang, & Cao, 2015) that allow GPGPU processing within the popular Spark parallel computing framework (Zaharia, Chowdhury, Franklin, Shenker, & Stoica, 2010) so that the high instruction throughput of GPGPUs can be combined with the scalability, ease of use and fault-tolerance of the 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. (Chen & Ng, 2004), (Chen, Ozu, & Oria, 2005), and (Ranu, Deepak, Telang, Deshpande, & Raghavan, 2015), very little research has been done in this area for GPGPUs. Recently, an algorithm called TKSimGPU (Leal, Gruenwald, Zhang, & You, 2015) 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 performing 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-Pruning, 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 to reduce the size of the candidate set, thus helping 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 (Zheng, Xie, & Ma, 2010).

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

2. Background and Related Work

In this section we first present the background material with the definitions of a trajectory,
Hausdorff distance, near-join trajectory similarity queries and top-K trajectory similarity queries. Then, we present related work. Before proceeding we present a summary of the notation used in this paper in Table 1.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>A query trajectory</td>
</tr>
<tr>
<td>( q )</td>
<td>A database trajectory</td>
</tr>
<tr>
<td>( P )</td>
<td>The set of query trajectories</td>
</tr>
<tr>
<td>( Q )</td>
<td>The set of database trajectories</td>
</tr>
<tr>
<td>( K )</td>
<td>The ( K ) parameter of top-K queries</td>
</tr>
<tr>
<td>( d(x,y) )</td>
<td>The Euclidean distance between points ( x ) and ( y )</td>
</tr>
<tr>
<td>( MBR(p) )</td>
<td>The minimum bounding rectangle of trajectory ( p )</td>
</tr>
<tr>
<td>( m_{p,q} )</td>
<td>The min-distance between the MBR of ( p ) and the MBR of trajectory ( q ). In other words: ( m_{p,q} = \min_{x,y \in MBR(p)} d(x,y) )</td>
</tr>
<tr>
<td>( M_{p,q} )</td>
<td>The max-distance between the MBR of ( p ) and the MBR of trajectory ( q ). In other words: ( M_{p,q} = \max_{x,y \in MBR(p)} d(x,y) )</td>
</tr>
<tr>
<td>( hausd(p,q) )</td>
<td>The Hausdorff distance between trajectory ( p ) and trajectory ( q )</td>
</tr>
<tr>
<td>( C )</td>
<td>The set of candidate pairs, as generated by a technique like TKSimGPU, that is a subset of ( P \times Q )</td>
</tr>
<tr>
<td>( C_p )</td>
<td>The subset of ( C ) consisting of all pairs that have ( p ) as its left component</td>
</tr>
</tbody>
</table>

### Table 1. Notation

#### 2.1 Background

**Definition of a Trajectory.** Given a set \( \{(x_i,y_i,t_i)|t_i \leq t_{i+1}, 1 \leq i < n\} \) 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 integer \( i \in \{1,...,n\} \) and such that \( \tau(x) \), with \( x \in [t_i,t_{i+1}] \), is the interpolated value between \( \tau(i) \) and \( \tau(i+1) \) (Cao, Wolfson, & Trajcevski, 2006).

**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 \( \max_{q \in Q} \min_{p \in P} d(p,q) \) (Nutanong, Jacox, & Samet, 2011).

**Definition of Near-join Trajectory Similarity Query.** Given a real number \( \epsilon > 0 \), two finite non-empty sets of trajectories \( P \) (the query set) and \( Q \) (the database), and a similarity measure \( \sigma: P \times Q \rightarrow \mathbb{R} \), a top-K trajectory similarity query returns for every \( p \) in \( P \) a set \( R_p \subseteq Q \) satisfying that for every \( q \in R_p \) it is the case that \( \sigma(p,q) \geq \epsilon \) (Keogh & Ratanamahatana, 2005).

**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 \( \sigma: P \times Q \rightarrow \mathbb{R} \), a top-K trajectory similarity query returns for every \( p \) in \( P \) a set \( R_p \subseteq Q \) satisfying that for every \( q \in R_p \) and \( q_{other} \in Q - R_p \) it is the case that \( \sigma(p,q_{other}) \leq \sigma(p,q) \) (Ding, Trajcevski, & Scheuermann, 2008).

### 2.2 Related Work

The works ERP (Chen & Ng, 2004), EDR (Chen, Özuş, & Oria, 2005), wDF (Ding, Trajcevski, & Scheuermann, 2008) and EDwP (Ranu, Deepak, Telang, Deshpande, & Raghavan, 2015) propose new trajectory similarity metrics and serial algorithms for trajectory similarity queries. Unlike these previous works, our technique, Top-KaBT, is a parallel algorithm to reduce 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 (Nutanong, Jacox, & Samet, 2011).

The Hausdorff distance arises naturally from certain problems like the bus route comparison problem, in which a transportation authority wants to replace one route by another, and the goal is to minimize the worst-case walking distance from a stop in the old route to its nearest stop in the new route (Nutanong, Jacox, & Samet, 2011). This distance has the advantage that it can be easily extended to mitigate the impact of noisy measurements (Nutanong, Jacox, & Samet, 2011). Another advantage of the Hausdorff distance is that it is 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. This means that the triangular inequality cannot be used to reduce the amount of work performed when processing trajectory similarity queries. For this reason, these techniques cannot be easily modified to be used with spatial data structures like R-trees, for which there already are parallel algorithms that work on GPUs (Zhang, You, & Gruenwald, 2013). The work UTGrid (Ma, Lu, Shou, & Chen, 2013) answers top-K trajectory similarity queries (KSQ) on uncertain trajectories with serial algorithms. The disadvantage of UTGrid is that, like the techniques previously discussed, it uses serial ad hoc indexing and query processing algorithms specially designed to address trajectory uncertainty.

All the previously mentioned 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 (Zhang, You, & Gruenwald, 2012) uses the Hausdorff distance as a similarity measure for near-join trajectory similarity queries on urban trajectories, while the work in (Gowanlock & Casanova, 2014) and (Gowanlock & Casanova, 2016) uses the Euclidean distance for the same type of query when applied to the study of moving galaxies and astrobiology.

Another parallel technique for trajectory similarity queries is TKSimGPU (Leal, Gruenwald, Zhang, & You, 2015), 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 candidate pairs. Such a set of candidates could be too big to fit into the GPGPU’s memory, making this technique not suitable for Big Data applications. Our proposed technique, Top-KaBT addresses this issue with its pruning strategies that specifically reduce the size of the candidate sets, and that also reduce the time wasted when computing the Hausdorff distance between spurious candidate pairs \((p,q)\).

Unlike all the works previously discussed in this section, our technique, Top-KaBT, is not a similarity measure. Instead it is a pruning algorithm designed to reduce the size of the candidate sets generated by techniques that compute trajectory similarities using the Hausdorff distance. Nonetheless, the pruning ideas behind Top-KaBT are valid not only for the Hausdorff distance, but also for any trajectory similarity measure that satisfies the properties of a metric. An example of such a parallel algorithm is TKSimGPU (Leal, Gruenwald, Zhang, & You, 2015). To accomplish its goal, Top-KaBT calculates the 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 3.2. 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.

3.2 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 3.9, which states that if we have a trajectory \(p\) with candidate pairs \(C_p = \{(p,q_0), (p,q_1), \cdots, (p,q_{n_p-1})\}\) sorted by the lower bounds to their respective Hausdorff distances, then if we find an integer \(0 \leq v_K \leq n_p - 1\) meeting certain conditions explained later, we will know that the \(K\) most similar trajectories to \(p\) will be among \(C_p = \{(p,q_0), (p,q_1), \cdots, (p,q_{v_K})\}\), and we can prune the remaining elements \(C^*_p = \{(p,q_{v_K+1}), (p,q_{v_K+2}), \cdots, (p,q_{n_p-1})\}\).

**Lemma 3.1**. For any \((p,q) \in P \times Q\), it is true that \(m_{p,q} \leq \text{hausd}(p,q) \leq M_{p,q}\).

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

**Definition 3.2** (Cut point set). Given the candidate set \(C_p = \{(p,q_0), (p,q_1), (p,q_2), \cdots, (p,q_{n_p})\}\) satisfying \(m_{p,q_i} \leq m_{p,q_{i+1}}\) 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.

3. THE TOP-KAAPT ALGORITHM

3.1 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
Example 3.3. If we have the following set of candidate pairs \( C_p = \{(p, q_0), (p, q_1), (p, q_2), (p, q_3)\} \) 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 \( C_p = \{(1,2)\} \) because \( M_{p,q} = 2.7 \leq 3.3 = m_{p,q_2} \), and \( M_{p,q_1} = 4.0 \leq 4.1 = m_{p,q_3} \).

Definition 3.4 (min-cut point). Given the candidate set \( C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p})\} \) satisfying \( m_{p,q_i} \leq m_{p,q_{i+1}} \) for \( 0 \leq i < n_p - 1 \) with cut-point set \( C_p \neq \emptyset \), the min-cut point of \( C_p \) is defined to be the \( K \)-th smallest element in \( C_p \).

Example 3.5. In Example 3.3 the min-cut point is \( \min C_p = 1 \).

Definition 3.6 (min-K-Cut point). Given the candidate set \( C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p})\} \) with \( m_{p,q_i} \leq m_{p,q_{i+1}} \) for \( 0 \leq i < n_p - 1 \) with cut-point set \( C_p \neq \emptyset \), the min-K-cut point of \( C_p \) is defined to be the K-th smallest element in \( C_p \).

Example 3.7. In Example 3.3 the min-K-cut point for \( K = 2 \) is 2.

Theorem 3.8. If \( v \) is a cut point of the following candidate set \( C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p})\} \) 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, \( m_{p,q_i} \leq m_{p,q_{i+1}} \) is true, and since \( m_{p,q_{i+1}} \) is a lower-bound of \( \text{hausd}(p, q_{i+1}) \), and \( M_{p,q_{i+1}} \) is an upper bound of \( \text{hausd}(p, q_{i+1}) \), then the following inequality holds \( \text{hausd}(p, q_v) \leq M_{p,q_{v}} \leq \text{hausd}(p, q_{v+1}) \). By induction, we can easily prove that \( \text{hausd}(p, q_v) \leq \text{hausd}(p, q_j) \) 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 3.9. If \( v_K \) is a min-K-cut point of the candidate set \( C_p = \{(p, q_0), (p, q_1), \cdots, (p, q_{n_p})\} \), 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 \( \{q_i | 1 \leq i \leq v_k\} \). We also know that, by the definition of min-K cut point, \( v_k \leq v_{k+1} \), and also that \( \text{hausd}(p, q_{v_k}) \leq \text{hausd}(p, q_j) \) for \( k+1 \leq j < n_p \). This implies that the \( k+1 \) nearest neighbors of \( p \) are in the set \( \{q_i | 0 \leq i \leq v_{k+1}\} \), which is what we wanted to prove.

Example 3.10. Continuing with Example 3.3 and using Theorem 3.9, we know that the top-2 nearest neighbors of trajectory \( p \) are contained in the set \( C_p = \{(p, q_0), (p, q_1), (p, q_2)\} \). This theorem allows us to discard the candidate pair \((p, q_3)\).

Example 3.11. In Figure 2 we have an object \( p \) and five objects \( q_0, q_1, q_2, q_3, q_4 \) located in a single-dimensional space generated by the vector \( X \). All objects are shown as circles. For each object \( q_i \), we have a lower bound for the distance from \( p \) to \( q_i \),...
denoted by $\text{LowerBound}[i]$ in the figure. Similarly, for each object $q_i$, we have an upper bound for the distance from $p$ to $q_i$, denoted by $\text{UpperBound}[i]$ in the figure. It can be seen that the array of objects $[q_0, q_1, q_2, q_3, q_4]$ satisfies $\text{LowerBound}[i] \leq \text{LowerBound}[i+1]$, so that the antecedent of Theorem 3.9 holds in this case. Additionally, it can be seen that $\text{UpperBound}[1] \leq \text{LowerBound}[2]$, so that, by definition, 1 is a cut-point of the candidate set in the figure. This means that the farthest that $q_1$ could possibly be from $p$ (the upper bound to its distance to $p$) is smaller than the closest that $q_2$ could be to $p$ (the lower bound to its distance to $p$). Therefore, we know for sure that objects $q_2, q_3$, and $q_4$ must be farther away from $p$ than $q_0$ and $q_1$, without explicitly computing the distances from $p$ to all the $q_i$'s. So, according to Theorem 3.9, if we are searching for the $K=1$ nearest neighbor to $p$, we only need to search in the set $\{q_0, q_1\}$.

Analogously, we have that $\text{UpperBound}[3] \leq \text{LowerBound}[4]$, which means that 3 is a 2-cut point of the candidate set in the figure. Therefore, according to Theorem 3.9, if we seek for the $K=2$ nearest neighbor to $p$, we only need to search in the set $\{q_0, q_1, q_2, q_3\}$ because, for sure, we know that $q_4$ is going to be farther away from $p$ than $q_0, q_1, q_2$, and $q_3$.

**Observation 3.12.** 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 (Ramaswami, Rastogi, & Shim, 2000): $\text{mindist}(R, S) = \sqrt{d_x^2 + d_y^2}$, 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 $\text{maxdist}(R, S) = \sqrt{c_x^2 + c_y^2}$, where $c_i = r'_i - p_i$ if $p_i < (r_i + r'_i)/2$, and $c_i = p_i - r'_i$ otherwise.

**Observation 3.13.** Given a candidate set $C_p = \{(p, q_0), (p, q_1), \ldots, (p, q_{n_p-1})\}$. Observation 3.12 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 $(p, q_i) \in C_p$.

3.3 DESCRIPTION OF TOP-KA$\text{B}T$'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, which is based on Theorem 3.9. The pseudocode algorithm for the pruning technique is in Figure 2, while Figure 3 and Figure 4 provide an illustrated example.

The main function is called SORT_PRUNING and is presented in Line 1 of Figure 2. This function 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 3.9. This function takes the 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 \((\text{low}_p\) and \(\text{upp}_p\), respectively) of the trajectory similarity between \(p\) and \(q\), using Observation 3.13. This is illustrated in Step 1 in Figure 3, where we can see that different thread blocks are assigned to 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 \(Q_p\), \(\text{low}_p\), and \(\text{upp}_p\), using the entries in \(\text{low}_p\) as keys; in this way we ensure that the premise of Theorem 3.9 is satisfied. An example of this is shown in Step 2 in Figure 3, where we see that the pairs corresponding to the first thread block have been sorted according to their lower bounds so that \((p_3,q_7)\) has smaller lower bound (whose value is 1.3) than \((p_3, q_1)\), 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 Figure 2, \(\text{low}_p\) is shifted 1 entry to the left for memory coalescing in line 23. The reason for this is that, according to Theorem 3.9, we test if \(M_{p,q_i} \leq m_{p,q_{i+1}}\) for every \(p\) and \(q_i\), so the value \(\text{low}_p[0]\) corresponding to \(m_{p,q_0}\) is never used. Figure 3 shows the left-shifting of the lower bounds array in Step 3. 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 3.9, 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 Steps 4 and 5 in Figure 3 and Figure 4, respectively.

The function HAUSDORFF_BOUNDS in Line 15, shown in Step 1 in Figure 3, receives a trajectory \(p\), and a list \(Q_p\) with the associated \(q\) trajectory candidates, and finds \(\text{low}_p\) and \(\text{upp}_p\) that satisfy: \(\text{low}_p \leq \text{Hausd}(p,q) \leq \text{upp}_p\). In Lines 17 to 20 \(\text{low}_p\) and \(\text{upp}_p\) are computed in parallel for every \(q\) in \(Q_p\) using Observation 3.13. This function exploits the memory coalescing unit 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.

The function FIND_CUT_POINT in Line 23 receives as input parameters a $p$ in $P$, an integer $K$, and the two arrays $\text{low}_p$ and $\text{upp}_p$ of the lower and upper bounds, respectively, and is in charge of finding the smallest $K$-cut point using Theorem 3.9. After the parallel loop in Lines 24 through 30, an array $\text{cut}_p$ is obtained, which is shown in Figure 3 Step 4. There we see that the $\text{cut}_p$ 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 $\text{cut}_p$ 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 inclusive prefix sum (Harris, Sengupta, & Owens, 2007) over $\text{cut}_p$ (which is the portion of the $\text{cut}_p$ array corresponding to $p$) is performed to obtain the array $\text{Pfx\_cut}_p$ of Line 31; this is shown in Figure 3 Step 4 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 $\text{Pfx\_cut}_p[i] \geq K$. 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 $\text{Pfx\_cut}_p$ portion of the second thread block at position 1. This function does memory coalescing because threads with consecutive indexes access adjacent memory locations in the $\text{cut}_p$ array. Also, all threads perform the same amount of work.

The function REMOVE in Line 35 receives as input parameters the array $\text{candidates}$ with the candidate trajectory pairs $(p, q)$, and an array $\text{cut}_pts$ of length $|\Pi_p(\text{candidates})|$ (where $\Pi_p(\text{candidates})$ is the projection on the left component ($P$) of the tuples in $\text{candidates}$). This last array satisfies that $\text{cut}_pts[i]$ is the cut point associated with the $i$-th $p$ trajectory in $\Pi_p(\text{candidates})$. Lines 37 through 43 create an array $B$ that contains the elements of $\text{cut}_pts + \text{offset} + 1$ in its even-indexed entries, and the elements of $|\{(p_j, q) \in \text{candidates} | p_j < p_i\}|$ in its odd-indexed entries. In Figure 4 we see in Step 5 that the elements of $\text{cut}_pts + \text{offset} + 1$ are $B[1]=2, B[3]=5$ and $B[5]=10$. The idea behind creating $B$ is to count how many pairs $(p, q)$ are going to be preserved for every $p$. In these same Lines 37 to 43, we create another array $\text{Alter\_1s0s}$ with 1s in its even entries and 0s in its odd entries. This array is used for run-length decoding (Fang, He, & Luo, 2010). Then Line 44 performs a parallel reduction to compute the array $\text{Counts}$ satisfying that $\text{Counts}[i] = B[2*i+1] - B[2*i]$. $\text{Counts}[2*i+1]$ is the number of $q$ candidates associated with $p_i$ that can be pruned away, while $\text{Counts}[2*i]$ indicates the number of $q$ candidates associated with $p_i$ that cannot be pruned away. In

---

Figure 4. Example run of the sort pruning algorithm of Top-KaBT (Steps 5 and 6)
Figure 4 we see that in Step 5 Counts[0] = 2 because B[1] − B[0] = 2−0 = 2, and Counts[1] = 1 because B[2] − B[1] = 3−2 = 1. This means that 2 pairs associated with p3 (which is the 0-th p candidate) cannot be pruned away, but 1 pair can be pruned away. Line 45 performs a run-length decoding over Counts (containing the counts of how many times the elements will occur in the final result of the run-length decoding) and Alter_Idxs (containing the elements that will be in the result of the run-length decoding); this is to obtain the array Stencil of length \( |\Pi_p(candidates)|\), 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 3.9.

We then create a new array Pruned of length equal to the sum of all the elements in Stencil. Lines 47 to 51 prune the spurious candidate pairs from candidates by writing into Pruned only those elements of candidates located at positions \( i\) such that Stencil\( [i]\) = 1. In Figure 4 Step 6 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.

### 3.4 Complexity of Top-KaBT

In this subsection we discuss the worst-case work and space complexity of the Top-KaBT pruning algorithm.

We now estimate the total amount of work performed by Top-KaBT in the function HAUSSDORFF_BOUNDS in Line 15 in Figure 2. Because this function computes the lower and upper bounds for the Hausdorff distance between \( p \) and \( q \) for every \((p,q)\) candidate pair in \( C \), and since we know that, according to Observation 3.12, the calculation of these lower and upper bounds has worst-case constant work complexity; therefore, the total amount of work done by the HAUSSDORFF_BOUNDS function has worst-case work complexity \( O(|C|) \) and worst-case space complexity \( O(|C|) \). Because the function FIND_CUT_POINT in Line 23 visits each candidate pair in \( C \) once, and in each visit it performs a constant amount of work, then the work complexity is \( O(|C|) \) to find all the 1-cut points in Lines 24 to 30, and \( O(|C|) \) amount of work to perform both the parallel prefix sum in Line 31, and to find the minimum in Line 32. So the total amount of work performed in the function FIND_CUT_POINT is \( O(|C|) \). The space complexity of this function is \( O(|C|) \).

In function REMOVE_CANDIDATES, we see that in Lines 37 to 43 of Figure 2 the total amount of work is again \( O(|C|) \) because the algorithm performs multiple passes over the array of candidates, doing constant work at each entry of this array. Then, in Lines 44 to 46, the total amount of work is \( O(|C|) \) because the parallel algorithm to find the adjacent differences, to perform run-length decoding, and to perform prefix sum have \( O(|C|) \) worst-case work complexity. In a similar fashion, Lines 47 to 51 have \( O(|C|) \) worst-case work complexity because the instructions at lines simply require writing a 1 or a 0 for every candidate pair in \( C \).

Examining now the function SORT_PRUNING, we see that it requires sorting \( C_p \) for every \( p \) in the query set \( P \), which requires \( O(|P| \cdot |C_p| \cdot \log(C_p)) \) amount of work to sort the each of the candidate sets \( C_p \). SORT_PRUNING then eventually calls the functions HAUSSDORFF_BOUNDS, FIND_CUT_POINT, and REMOVE_CANDIDATES, whose total amount of work have already been calculated. We conclude then that the overall worst-case work complexity of Top-KaBT is \( O(|C| + |P| \cdot |C_p| \cdot \log(C_p)) \), and the worst-case space complexity is also \( O(|C|) \).

### 4. Experiments and Results

In this section we describe the dataset, the hardware and software environment, and the experiments used to compare the state of the art top-K trajectory similarity query processing algorithm on GPGPU, TKSimGPU (Leal, Gruenwald, Zhang, & You, 2015), 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 \( p \in P \).

### 4.1 Datasets and Experiment Setup

For our experiments we use the Microsoft GeoLife trajectory dataset (Zheng, Xie, & Ma, 2003) 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 with 4GB of RAM. Our code uses Thrust 1.8 (Hoberock & Bell, 2010), CUDA 7.0, and was compiled with g++ with O2 optimizations.

4.2 EXPERIMENTS

We now describe three experiments to evaluate the impacts of the size of the query set, the size of the database, the size of the parameter \(K\) on the average query processing time of TKSimGPU combined with Top-KaBT (denoted in this section as TKSimGPU + Top-KaBT), TKSimGPU, and the naïve exhaustive GPGPU search algorithm (denoted by naïveGPU). NaïveGPU is a parallel GPGPU algorithm that works by computing the Hausdorff distance between \(p\) and \(q\) for every \((p,q)\) in \(P \times Q\), then sorting the pairs inside each set \(C_p\) in increasing order of Hausdorff distance, finally taking for every \(C_p\), the \(K\) pairs with smallest Hausdorff distance. We also study the impact of these three parameters on the percentage of candidate pairs \((p,q)\) in the set \(P \times Q\) whose similarity is computed by each algorithm. To illustrate this concept of the percentage of candidate pairs explored, notice that naïveGPU always explores (computes the Hausdorff distance between) 100% of the candidate pairs in \(P \times Q\) because by its own nature, naïveGPU performs an exhaustive search on all possible candidate pairs. Therefore, the lower this percentage, the more efficient the pruning technique is since it computes the similarity measure on a smaller subset of \(P \times Q\). In all the experiments, we choose a grid cell size of 128 \(\times\) 128, and for TKSimGPU+Top-KaBT and TKSimGPU we take a sample size of 512 elements and use 512 threads per block. 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 10,330,000), and \(K=70\). We vary the query set size from 20 to 100 trajectories (up to 17,000 points \((x,y,t)\)).

In Figure 5 we see that the average query execution times of all three techniques seem linear. This is because the average query execution time is dominated by the average number of \((p,q)\) candidates that remain before running 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 is 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 function of the size of \(P\).

In Figure 5 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. This is because, as we can see in Figure 6, the candidate set size of TKSimGPU+Top-KaBT is on average 4 times smaller than the one that TKSimGPU alone generates. TKSimGPU is also 11 times faster than naïveGPU because its candidate set size is 15 times smaller than the naïve’s.

Figure 6 shows the impact of the size of the query set (|P|) on the percentage of pairs P×Q explored (i.e., the percentage of pairs that have their Hausdorff distances computed). In this figure we observe that naïveGPU always explores 100% of the pairs in P×Q, as expected. In Figure 6 we also observe that for all three algorithms the percentage of (p,q) candidate pairs in P×Q pruned does not seem to depend on the size of the query set. In particular, TKSimGPU+Top-KaBT does not show a strong dependency on the size of the query set P. The reason for this is that each query trajectory p in P has an approximately equal number of (p,q) candidate pairs pruned; therefore, by increasing the size of the query set P by a factor of n times leads to an n time increase of the number of candidate pairs in P×Q, but the number of candidate pairs pruned also increases by n (because Theorem 3.9 prunes the same number of pairs for every p in P), which implies that the percentage of candidate pairs pruned is nearly constant, which is what we observe in Figure 6.

The previous observation is also consistent with Figure 5, in which we saw a linear relationship between |P| and the average query execution time. This is because the percentage of candidates pruned remains constant as the query set size increases, so the amount of non-pruned pairs (which is proportional to the average query processing time) must also increase linearly with |P|.

In Figure 7 we observe the impact of the size of the query set (|P|) on the average execution time of the pruning algorithm Top-KaBT alone (without counting the execution time of TKSimGPU). We observe that as the size of the query set increases, the average execution time for this pruning algorithm increases. However, comparing the execution times in Figure 5 and Figure 7 we observe that the average query execution time of just the Top-KaBT portion of TKSimGPU + Top-KaBT represents around 2.5% of the total average execution time of TKSimGPU+TopKaBT. This implies that the overhead of adding the Top-KaBT pruning on top of TKSimGPU is small in comparison with the execution time of TKSimGPU alone.

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

In Figure 8 we observe that the average query execution time for the three techniques seems to be a linear function 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 Figure 8 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. In this figure we observe also that TKSimGPU is 13 times faster than naïveGPU because naïveGPU computes P×Q, while TKSimGPU performs pruning and thus reduces the size of the candidate pairs set.

Figure 9 shows the impact of the database size (|Q|) on the percentage of candidate pairs in P×Q that are exhaustively searched in the refine stage. We also see that the percentage of candidate pairs pruned by Top-KaBT initially decreases with the size of the database. This behavior is expected of Top-KaBT because increasing the size of the database can either decrease or increase the value of K-cut points. To see this, assume K=1, a fixed query trajectory p, a fixed database Q, and such that the candidate pairs associated with p are q₀, q₁, q₂ with lower bounds (for their respective Hausdorff distances to p) 1, 2 and 4, respectively, and with upper bounds (for their respective Hausdorff distances to p) 3, 3, and 5, respectively. Then, by definition of a cut point, 1 is a cut point associated with the candidate set of p. Now, consider another trajectory q₄ in the database with lower bound 2.5 and upper bound 5. If q₄ is added to the set of candidate pairs of p, then this would increase the cut point to 3, so no candidate pairs are pruned. But if a trajectory q₅ with lower bound (for its distance to p) 0.5 and upper bound 0.75 is added instead of q₄ to the set of candidate pairs of p, then the cut point associated would decrease to 0, which would increase the percentage of candidate pairs pruned. Therefore, the way that increases in the database size impact the percentage of candidate pairs pruned depends on the spatial distribution of the dataset.

In Figure 10 we observe a similar behavior to the one in Figure 7, where the average execution time for the Top-KaBT pruning portion increases with the size of the database. Again, we confirm that the execution time of the Top-KaBT portion represents, on average, only 2.5% of the total execution time of TKSimGPU+Top-KaBT, so Top-KaBT adds very little overhead to the execution time of TKSimGPU.

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 Figure 11 we observe that the average query execution time 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×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 Figure 11 we...
observe that TKSimGPU+Top-KaBT outperforms TKSimGPU in terms of average query processing time, and TKSimGPU outperforms naïveGPU. This is because, again, the average query processing time is dominated by the size of the candidate pairs set.

In Figure 12 we see the impact of $K$ on the percentage of candidate pairs pruned by each of the techniques compared. In particular, this figure shows that the percentage of candidate pairs in $P \times Q$ explored by TKSimGPU+Top-KaBT increases with $K$. This is because the set of all possible candidate pairs $P \times Q$ is fixed, so for a given query trajectory $p$ in $P$, a linear increase in $K$ forces Top-KaBT to find $K$-cut points further along to the end of the array of candidates, which means that more candidate pairs are produced as a result of this. From this figure we can also observe that the size of the candidate pair set of TKSimGPU+Top-KaBT is on average 5 times smaller than the size of the candidate pair set of TKSimGPU, which in tum is 4 times smaller than that of naïveGPU.

In Figure 13 we observe that the average execution time of Top-KaBT exhibits on overall tendency to increase as $K$ grows bigger. However, its behavior looks less like a straight line than in the case of Figure 7 and Figure 10. The reason for this is that when the value of $K$ is changed and the sizes of the query set and the database stay constant, much of the work performed by Top-KaBT remains the same. For example, the calculation of the lower and upper bounds to the Hausdorff distances (Line 4 of Figure 2), the sorting of $Q_p$ (Line 5 of Figure 2), the left shifting of the array (Line 6 of Figure 2), and the finding of the cut-points (Line 7 of Figure 2), etc. require the same amount of work if the query set and the database sizes are kept invariant. The only difference in the amount of work that is introduced by changing $K$ comes at Lines 47 to 51 in Figure 2 when the spurious candidate pairs are removed. Since larger values of $K$ usually lead to a lower percentage of candidate pairs pruned, then larger values of $K$ require more write operations in Line 49, which lead to slightly longer execution times, as can be seen in Figure 13.

5. CONCLUSIONS AND FUTURE WORK
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 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, an exhaustive search technique on GPGPU, 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. Additionally, in the experiments we observed that the time overhead incurred by Top-KaBT is very small because it only represents 2.5% of the average query execution time spent by TKSimGPU when combined with Top-KaBT.

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 1 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.

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

7. REFERENCES


Gowanlock, M., Casanova, H. (2016). Distance threshold similarity searches: Efficient Trajectory Indexing on the GPU. In IEEE Transactions on Parallel and Distributed Systems (TPDS), 27(9), (pp. 2533–2545).


Authors

Eleazar Leal is working for his Ph.D. in Computer Science at the University of Oklahoma. He received his MS in Computer Science from the University of Oklahoma, and BS in Computer Engineering from the Universidad Simón Bolívar in Venezuela. His research interests are: trajectory databases, trajectory data mining and GPU computing.

Le Gruenwald is a Professor, Dr. David W. Franke Professor, and Samuel Roberts Noble Foundation Presidential Professor in the School of Computer Science at The University of Oklahoma. She received her Ph.D. in Computer Science from Southern Methodist University, MS in Computer Science from the University of Houston, and BS in Physics from the University of Saigon. She worked for National Science Foundation as a Cluster Lead and Program Director of the Information Integration and Informatics program and a Program Director of the Cyber Trust program. Dr. Gruenwald’s major research interests include Stream and Sensor Databases, Cloud Databases, Multimedia Databases, Distributed Databases, Mobile Databases, Object-Oriented Databases, Real-Time Databases, Web Databases, Data Mining, and Data Warehouse.

Jianting Zhang is an Assistant Professor in Geographical Information Systems in the Department of Computer Science of the City College of New York. He received his Ph.D. in Computer Science from the University of Oklahoma. His research interests are: High-Performance Geospatial Computing, Environmental Cyberinfrastructure, Spatial Databases and GIS Applications, Geospatial Visual Analytics, Multispectral and Hyperspectral Remote Sensing Data Processing.