UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

PARALLEL PROCESSING OF TOP-K TRAJECTORY SIMILARITY QUERIES ON BIG DATA USING GPUS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

ELEAZAR ENRIQUE LEAL GONZALEZ
Norman, Oklahoma
2017

PARALLEL PROCESSING OF TOP-K TRAJECTORY SIMILARITY QUERIES ON BIG DATA USING GPUS

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Le Gruenwald, Chair

Dr. Sudarshan Dhall

Dr. Changwook Kim

Dr. S. Lakshmivarahan

Dr. Scott Moses

© Copyright by ELEAZAR ENRIQUE LEAL GONZALEZ 2017
All Rights Reserved.

TABLE OF CONTENTS

LIST OF FIGURES viii

LIST OF TABLES x

ABSTRACT xi

CHAPTER I INTRODUCTION 1

1 Objective 1

2 Background 1

2.1 Notation 2

2.2 Geometric Background 3

2.3 Trajectory Similarity Background 7

2.4 GPU Background 10

2.4.1 What are GPUs? 10

2.4.2 Why and when to use GPUs? 11

2.4.3 GPU Programming Model 12

2.4.4 GPU Issues 14

2.4.4.1 Low global memory bandwidth relative to the number of threads 14

2.4.4.2 Low PCIe memory bandwidth 16

2.4.4.3 Efficient use of shared memory banks 17

2.4.4.4 Thread divergence 17

2.4.4.5 Load balancing 18

3 General Issues of Top-K Trajectory Similarity Query Processing Techniques 19

3.1 Different trajectory sizes 19

3.2 Local time shifts 20

3.3 Measurement uncertainty 22

3.4 Model uncertainty 23

3.5 Triangular inequality 25

3.6 Inter-trajectory sampling rate variation 26

3.7 Intra-trajectory sampling rate variation 28

3.8 Sampling phase variation 30

3.9 Dimensionality of the manually-tuned parameter space 31

3.10 Large Databases and Large Trajectory Sizes 32

4 Contribution 33

5 Organization 36

CHAPTER II LITERATURE REVIEW 38

1 Literature Review of Top-K Trajectory Similarity Query Processing Techniques 38

1.1 Geometry-based techniques 38

1.1.1 Euclidean Distance Technique 38

1.1.2 Hausdorff Distance Technique 41

1.1.3 w-constrained Discrete Fréchet Distance (wDF) 44

1.1.4 DISSIM 48

1.2 Edit distance-based Techniques 52

1.2.1 Dynamic Time Warping (DTW) 52

1.2.2 Longest Common Subsequence (LCSS) 55

1.2.3 Edit Distance on Real Sequence (EDR) 57

1.2.4 Edit Distance with Real Penalty (ERP) 60

1.2.5 Edit Distance With Projection (EDwP) 62

1.2.6 MA 64
1.3 Probability-based techniques 67
1.3.1 KSQ 67
1.4 Feature Comparison of Top-K Trajectory Similarity Query Processing Techniques 71
2 Literature Review of Techniques for Estimating Uncertain Trajectories 75
2.1 Techniques that do not exploit a database of trajectories 75
2.1.1 Mean and Median Filter 76
2.1.2 Kalman Filter 77
2.1.3 Particle Filter 79
2.2 Techniques that exploit a database of trajectories 81
2.2.1 HRIS 81
2.2.2 Chazal et al.'s Algorithm 84
2.3 Feature Comparison of Techniques for Estimating Uncertain Trajectories 86
CHAPTER III PROPOSED SYSTEM AND TECHNIQUES 89
1 Overview of the proposed system and techniques 89
2 TKSimGPU: A GPU technique for Top-K Trajectory Similarity Query Processing 92
2.1 Motivation of TKSimGPU 92
2.2 Overview of TKSimGPU 93
2.3 The TKSimGPU Algorithm 94
2.4 Parallel query execution of top-K trajectory similarity queries on GPUs 99
3 The Top-KaBT Algorithm: A GPU Technique for Pruning Candidate Sets that Arise when Processing Top-K Trajectory Queries 103
3.1 Motivation of Top-KaBT 103
3.2 Overview of Top-KaBT 104
3.3 Theoretical Foundations of Top-KaBT's Pruning Strategy 105
3.4 Description of Top-KaBT's Pruning Strategy 110
4 The TrajEstU Algorithm: A GPU Technique for Reducing Trajectory Uncertainty when Processing Top-K Trajectory Queries 118
4.1 Motivation of TrajEstU 118
4.2 Overview of TrajEstU 118
4.3 Description of TrajEstU 119
4.3.1 Pre-processing stage 119
4.3.2 Model-Fitting Stage 120
4.3.2.1 Constant Acceleration Model 121
4.3.2.2 Incorporation of Trajectory Patterns 123
4.3.2.3 Selecting the Best Constant Acceleration Model 125
4.3.2.4 Variable Acceleration Model 126
4.3.2.5 Model Coupling Function 127
4.3.3 Trajectory Estimation Stage 128
4.4 Details of TrajEstU 129
4.4.1 Pre-processing Stage 129
4.4.2 Model-Fitting Stage 129
4.4.3 Trajectory Estimation Stage 130
5 TraclusGPU: A parallel GPU technique for local clustering of trajectories 133
5.1 Motivation of TraclusGPU 133
5.2 Overview of TraclusGPU 135
5.3 Description of TraclusGPU 136
5.3.1 Partitioning Stage 136

5.3.2 Local Trajectory Clustering Stage 137
5.3.3 Representative Trajectory Finding Stage 139

CHAPTER IV PERFORMANCE ANALYSIS 141
1 Theoretical Analysis 141
1.1 Complexity Analysis for TKSimGPU 141
1.2 Complexity Analysis for Top-KaBT 144
1.3 Complexity Analysis for TrajEstU 146
1.4 Complexity Analysis for TraclusGPU 147
2 Experimental Analysis 149
2.1 Experimental Analysis of TKSimGPU 149
2.1.1 Experimental Setup 149
2.1.1.1 Hardware and Software Description 149
2.1.1.2 Datasets and experiment setup 149
2.1.1.3 Competing Algorithms 150
2.1.1.4 Experimental Parameters 151
2.1.1.5 Performance Metrics 152
2.1.2 Experimental Results 152
2.1.2.1 Impact of the query set size 152
2.1.2.2 Impact of the database size 153
2.1.2.3 Impact of K 155
2.1.2.4 Conclusions of TKSimGPU's Experimental Results 156
2.2 Experimental Analysis of Top-KaBT 157
2.2.1 Experimental Setup 157
2.2.1.1 Hardware and Software Description 157
2.2.1.2 Datasets and experiment setup 157
2.2.1.3 Competing Algorithms 157
2.2.1.4 Performance Metrics 158
2.2.1.5 Experimental Parameters 158
2.2.2 Experimental Results 159
2.2.2.1 Impact of the query set size (|P|) 159
2.2.2.2 Impact of the database size (|Q|) 162
2.2.2.3 Impact of K 165
2.2.2.4 Conclusions of Top-KaBT's Experimental Results 168
2.3 Experimental Analysis of TrajEstU 169
2.3.1 Experimental Setup 169
2.3.1.1 Hardware and Software Description 169
2.3.1.2 Datasets and Experiment Setup 170
2.3.1.3 Competing Algorithms 171
2.3.1.4 Experimental Parameters 171
2.3.1.5 Performance Metrics 173
2.3.2 Experimental Results 174
2.3.2.1 Impact of the Sampling Rate 174
2.3.2.2 Impact of the Query Length 179
2.3.2.3 Impact of the Standard Deviation of the Noise 183
2.3.2.4 Impact of the Acceleration Tolerance 187
2.3.2.5 Impact of the Dataset Size 188
2.3.2.6 Conclusions of TrajEstU's Experimental Results 189
2.4 Experimental Analysis of TraclusGPU 191
2.4.1 Experimental Setup 191
2.4.1.1 Hardware and Software Description 191

2.4.1.2 Datasets and Experiment Setup 191
2.4.1.3 Competing Algorithms 191
2.4.1.4 Experimental Parameters 191
2.4.1.5 Performance Metrics 192
2.4.2 Experimental Results 192
2.4.2.1 Impact of the Dataset Size 192
2.4.2.2 Conclusions of TraclusGPU's experiment results 193

CHAPTER V CONCLUSIONS AND FUTURE WORK 195

1 Summary of the Performance Results 196
1.1 Summary of the Results of TKSimGPU 196
1.2 Summary of the Results of Top-KaBT 199
1.3 Summary of the Results of TrajEstU 201
1.4 Summary of the results of TraclusGPU 203

2 Future Research 205

REFERENCES 209

LIST OF FIGURES

Figure 1. Examples of MBRs..3
Figure 2. Examples of eMBRs..4
Figure 3. Example of a trajectory 5
Figure 4. Example of trajectory similarity..7
Figure 5. Example of top-K trajectory similarity query 9
Figure 6. GPU description 11
Figure 7. GPU memory space..14
Figure 8. GPU memory coalescing..16
Figure 9. Thread divergence 18
Figure 10. Different trajectory sizes 20
Figure 11. Local time shifts 21
Figure 12. Measurement/instrumentation uncertainty 23
Figure 13. Model uncertainty...24
Figure 14. Inter-trajectory sampling rate variation.......................................27
Figure 15. Intra-trajectory sampling rate variation.......................................29
Figure 16. Sampling phase variation 30
Figure 17. Workflow of our proposed system 91
Figure 18. Overall algorithm..91
Figure 19. TKSimGPU algorithm...95
Figure 20. Relationship between Top-KaBT and top-K trajectory similarity query processing algorithms 105
Figure 21. Example of K-cut point 108
Figure 22. Sort pruning algorithm of Top-KaBT...111
Figure 23. Example run of the sort pruning algorithm of Top-KaBT (Steps 1 and 2) .112
Figure 24. Example run of the sort pruning algorithm of Top-KaBT (Steps 3 and 4) .114
Figure 25. Example run of the sort pruning algorithm of Top-KaBT (Step 5)............115
Figure 26. Example run of the sort pruning algorithm of Top-KaBT (Step 6)............117
Figure 27. Finding representative trajectories 124
Figure 28. Timestamp calculation for a set of cluster points........................125
Figure 29. Model coupling function 127
Figure 30. TrajEstU pseudocode...132
Figure 31. Pseudocode of the TraclusGPU algorithm 140
Figure 32. Query set size vs. execution time (TKSimGPU).........................153
Figure 33. K vs. execution time (TKSimGPU)..154
Figure 34. Database size vs. execution time (TKSimGPU)..........................154
Figure 35. Query set size vs. execution time (Top-KaBT)...........................159
Figure 36. Query set size vs. % candidate pairs explored (Top-KaBT) 160
Figure 37. Query set size vs. execution time of Top-KaBT alone................161
Figure 38. Database size vs. query execution time.......................................162
Figure 39. Database size vs. % candidate pairs explored 163
Figure 40. Database size vs. execution time of Top-KaBT alone 164
Figure 41. K vs. query execution time...165
Figure 42. K vs. % candidate pairs explored 166

Figure 43. K vs. execution time of Top-KaBT alone ... 167
Figure 44. Sampling rate vs. accuracy (deer dataset) ... 175
Figure 45. Sampling rate vs. execution time (deer dataset) ... 176
Figure 46. Sampling rate vs. accuracy (hurricane dataset) ... 177
Figure 47. Sampling rate vs. execution time (hurricane dataset) ... 178
Figure 48. Sampling rate vs. accuracy (synthetic dataset) ... 179
Figure 49. Sampling rate vs. execution time (synthetic dataset) ... 179
Figure 50. Query length vs. accuracy (deer dataset) ... 180
Figure 51. Query length vs. execution time (deer dataset) ... 181
Figure 52. Query length vs. accuracy (hurricane dataset) ... 181
Figure 53. Query length vs. execution time (hurricane dataset) ... 182
Figure 54. Query length vs. accuracy (synthetic dataset) ... 182
Figure 55. Query length vs. execution time (synthetic dataset) ... 183
Figure 56. Standard deviation of the measurement noise vs. accuracy (deer dataset) ... 184
Figure 57. Standard deviation of the measurement noise vs. execution time (deer dataset) ... 185
Figure 58. Standard deviation of the measurement noise vs. accuracy (hurricane dataset) ... 185
Figure 59. Standard deviation of the measurement noise vs. execution time (hurricane dataset) ... 186
Figure 60. Standard deviation of the measurement noise vs. accuracy (synthetic dataset) ... 186
Figure 61. Standard deviation of the measurement noise vs. execution time (synthetic dataset) ... 187
Figure 62.Acceleration tolerance vs. accuracy (deer dataset) ... 187
Figure 63. Acceleration tolerance vs. accuracy (hurricane dataset) ... 188
Figure 64. Acceleration tolerance vs. accuracy (synthetic dataset) ... 189
Figure 65. Number of segments vs. execution time (synthetic dataset) ... 193

LIST OF TABLES

Table 1. Notation 3
Table 2. Feature comparison of top-K trajectory similarity query processing techniques
.... 74
Table 3. Experimental parameters of TKSimGPU and Top-KaBT 152
Table 4. Experimental parameters of TrajEstU 173
Table 5. Experimental parameters of TraclusGPU 192

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 volumes of the trajectory query sets and databases, along with their associated uncertainty, pose significant computational challenges. One way to address these challenges is through the use of parallel architectures like GPUs, and through the use of models that can produce accurate trajectory estimates. Nevertheless, not much research has been done to design efficient and scalable techniques to process this type of query on parallel architectures.

In this dissertation, we propose a novel system to process top-K trajectory similarity queries in parallel on Big Data using GPUs that is capable of handling both certain and uncertain trajectory data. The system consists of four novel algorithms: TKSimGPU to process top-K trajectory similarity queries; Top-KaBT to reduce the size of the candidate set generated by top-K trajectory similarity query algorithms; TrajEstU to estimate the true trajectory when data uncertainty exists; and TraclusGPU to perform local trajectory clustering to aid in the preprocessing stage of TrajEstU. TKSimGPU works by iteratively processing near-join similarity queries, while Top-KaBT calculates the lower and upper bounds of the Hausdorff distance between candidate pairs, and then

uses these bounds to remove spurious candidates. Top-KaBT exploits GPUs to improve TKSimGPU by ensuring load balancing across the threads, ensuring memory coalescing, and using special pruning techniques that reduce the size of the candidate set. TrajEstU splits the lifetime of an object's trajectory into time intervals where the object's acceleration is nearly constant. Then TrajEstU uses the local trajectory clusters to obtain the movement patterns that are prevalent in the areas where trajectories have low-sampling rates, and uses linear regression to fit a constant acceleration model to the observed positions of the moving object. Finally, TraclusGPU helps TrajEstU scalably find those local trajectory clusters that are used in the construction of trajectory models.

Extensive theoretical and experimental evaluations performed on our proposed techniques showed that each of them has better performance in terms of accuracy and execution time than state-of-the-art techniques when applied to large real-life and synthetic trajectory datasets for Big Data applications.

CHAPTER I
INTRODUCTION

1 Objective

The objective of this research is to develop a novel system to process top-K trajectory similarity queries in parallel on Big Data using GPUs that is capable of handling both certain and uncertain trajectory data that addresses the following characteristics:

- Support for trajectories of different sizes
- Support for intra-trajectory sampling rate variation
- Measurement uncertainty
- Model uncertainty
- Triangular inequality
- Scalability through parallel processing on GPUs

The remainder of this chapter is organized as follows. Section 2 presents the background material on trajectories, top-K trajectory similarity queries, and Graphics Processing Units (GPUs). Then Section 3 discusses the issues and challenges that arise when designing parallel top-K trajectory similarity query processing techniques.

2 Background

In this section we discuss the background concepts that are necessary in order to follow the ideas introduced in this dissertation. This section consists of four subsections: Section 2.1 describes the notation used in this work; Section 2.2 presents the geometric background concepts; Section 2.3 presents the concept of trajectory similarity; and

finally, Section 2.4 provides an introduction to GPUs, their programming model, and their issues.

2.1 *Notation*

We now present in Table 1 a summary of the notation used in this dissertation.

Notation	Meaning
p, q, r, s, p_i, q_i	Trajectories
$p[i]$	The i-th-point of trajectory p, where i is a positive integer.
$p[i].x$	The x-component of the i-th-point of trajectory p
$p[i].y$	The y-component of the i-th-point of trajectory p
$p[i].t$	The timestamp of the i-th-point of trajectory p
$\|p\|$	The number of points in trajectory p
$p[I]$	The set of points of trajectory p whose timestamps fall within the time interval $I=[t_0,t_f]$
P	The set of query trajectories
Q	The set of database trajectories
K	The K parameter of top-K queries
$d(x,y)$	The Euclidean distance between points x and y
$MBR(p)$	The minimum bounding rectangle of trajectory p
$EMBR(p,\varepsilon)$	The ε-extended minimum bounding rectangle of trajectory p
$m_{p,q}$	The min-distance between the MBR of p and the MBR of trajectory q. In other words, $m_{p,q} = \min_{x\in MBR(p),y\in MBR(q)} d(x,y)$.
$M_{p,q}$	The max-distance between the MBR of p and the MBR of trajectory q. In other words, $M_{p,q} = \max_{x\in MBR(p),y\in MBR(q)} d(x,y)$.
$hausd(p,q)$	The Hausdorff distance between trajectory p and trajectory q
C	The set of candidate pairs, as generated by a technique like TKSimGPU, that is a subset of $P\times Q$

C_p	The subset of C consisting of all pairs that have p as its left component. In other words: $C_p = \{(p, q) \in C \mid q \in Q\}$.
$M[i, j]$	The element of the i-th-row and j-th column of matrix M
$M[i, :]$	The i-th-row of matrix M
c.repr	The representative trajectory of the trajectory cluster c
5e3, 7e6	This is the *scientific e notation* and in these cases it denotes the values 5×10^3 *and* 7×10^6

Table 1. Notation

2.2 *Geometric Background*

2.2.1 Definition (Minimum Bounding Rectangle): Given any set of points in the plane, its *minimum bounding rectangle* (MBR) is the smallest rectangle that contains (bounds) such set. Figure 1 illustrates the concept of MBRs. In the left part of the figure there is a set points, and in the right part, there is another set of points (that form a trajectory). From the figure it is easy to see that their MBRs are the smallest rectangles containing each of the sets.

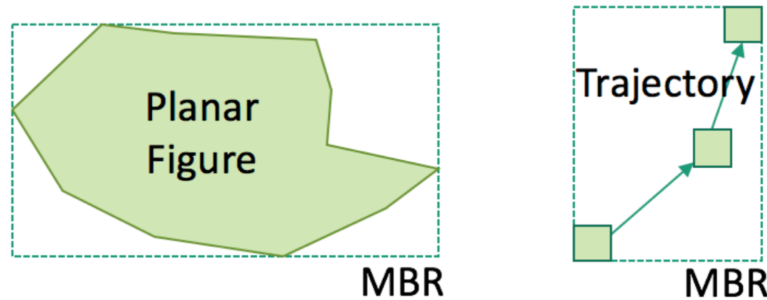

Figure 1. Examples of MBRs

2.2.2 Definition (ε-extended Minimum Bounding Rectangle): Given a real number $\varepsilon > 0$, and any set of points, its *ε-extended minimum bounding rectangle* (eMBR) is a

rectangle that results from extending each side of the MBR of the set of points by ε. So, if the MBR of a given set of points has upper left corner coordinates (u_x, u_y) and lower left corner coordinates (l_x, l_y), then the eMBR has upper left corner coordinates $(u_x - \varepsilon, u_y + \varepsilon)$ and lower left corner coordinates $(l_x + \varepsilon, l_y - \varepsilon)$. Figure 2 illustrates the concept of eMBRs. In the left part of the figure, there is a planar figure with its associated MBR, and in the right part, there is another set of points (that form a trajectory) with its associated MBR. In this figure we see that each of the MBRs has been extended in all directions by an amount ε.

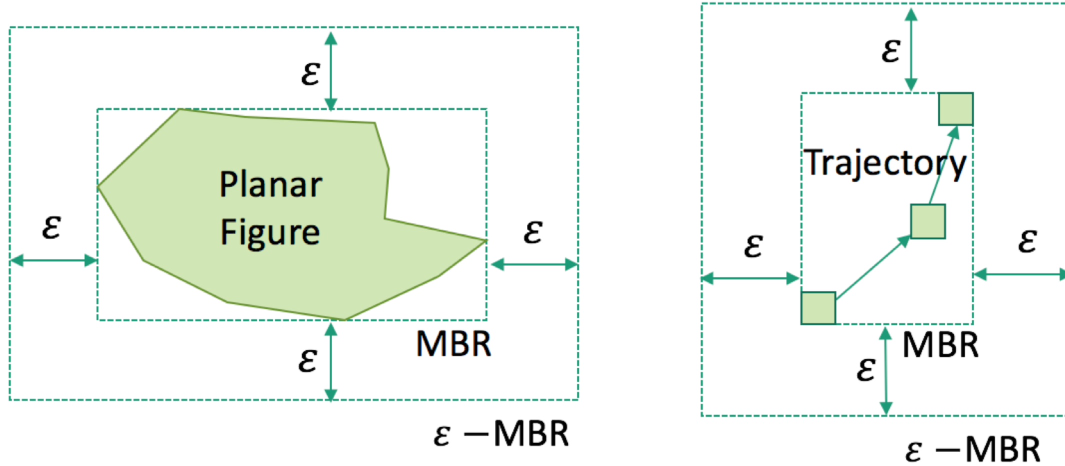

Figure 2. Examples of eMBRs

2.2.3 Definition (Trajectory): Informally, 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. More formally, 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

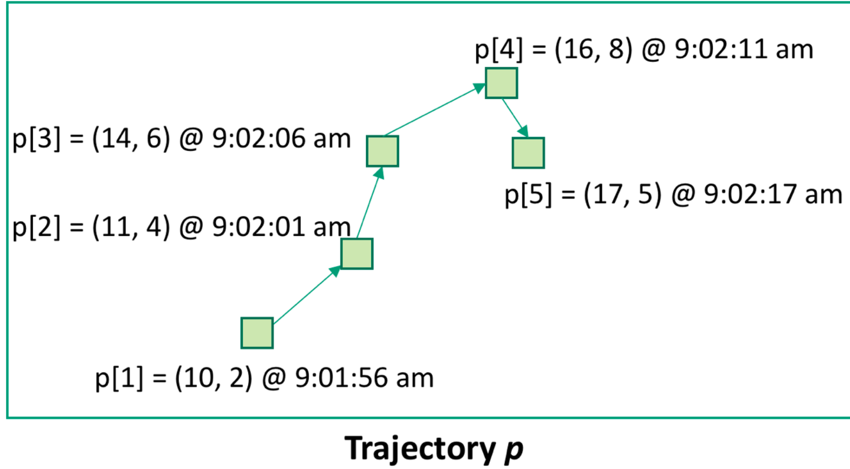

Figure 3. Example of a trajectory

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 $\tau(i + 1)$ [CW06]. Figure 3 shows an example trajectory with 5 points. At each point we use the notation $p_1 = (10, 2)$ @ 9:01:56 am to denote that the x coordinate of p_1 is 10, its y coordinate is 2, and its associated timestamp is 9:01:56 am.

2.2.4 Definition (Sub-trajectory): Given a trajectory $p = \{(x_i, y_i, t_i) | t_i \leq t_{i+1}, 1 \leq i < n\}$, we define the sub-trajectory of p during the interval *[a,b]*, denoted by *traj[a,b]*, with $a < b$, as the set of all points of the trajectory p with timestamps between a and b. More formally, it is the subset $\{(x_i, y_i, t_i) | a \leq t_i \leq t_i + 1 \leq b, 1 \leq i < n\}$.

2.2.4 Definition (Size of a Trajectory): Given a trajectory $p = \{(x_i, y_i, t_i) | t_i \leq t_{i+1}, 1 \leq i < n\}$, its *size* is the total number of points that belong to the trajectory. For example, the trajectory in Figure 3 has 5 points and is said to have size 5.

2.2.5 Definition (Length of a Trajectory): Given a trajectory $p = \{(x_i, y_i, t_i) | t_i \leq t_{i+1}, 1 \leq i < n\}$, its *length* is the summation of the distances between consecutive points in the trajectory. More formally,

$$Length(p) = \sum_{i=1}^{n-1} d(p[i], p[i+1])$$

For example, the trajectory in Figure 3 has length $d(p_1, p_2) + d(p_2, p_3) + d(p_3, p_4) + d(p_4, p_5)$.

2.2.6 Definition (Lifetime of a Trajectory): Given a trajectory $p = \{(x_i, y_i, t_i) | t_i \leq t_{i+1}, 1 \leq i < n\}$, its *lifetime* is $[t_0, t_{n-1}]$, i.e., the smallest closed time interval containing the projections of the points in *p* into the time domain. For example, the trajectory in Figure 3 has lifetime [9:01:56 am, 9:02:17 am].

2.2.7 Definition (Average Trajectory Sampling Rate): Given a trajectory $p = \{(x_i, y_i, t_i) | t_i \leq t_{i+1}, 1 \leq i < n\}$, its *sampling rate* is the average time elapsed between consecutive points in the trajectory. More formally,

$$Rate(p) = \frac{\sum_{i=1}^{n-1} t_{i+1} - t_i}{n-1}$$

For example, the trajectory in Figure 3 has an average trajectory sampling rate of (5s + 5s + 5s + 6s) / 4 = 21/4 s = 5.25s.

2.2.8 Definition (Low Sampling Rate Trajectory): Given a trajectory, it is said to be a *low-sampling rate trajectory* if the average time span between any two of its consecutive points is greater than a predefined threshold. A trajectory that is not a low-sampling rate trajectory is said to be a *high-sampling rate trajectory*.

2.3 *Trajectory Similarity Background*

2.3.1 Definition (Trajectory Similarity): The informal notion of trajectory similarity is as follows. Any two trajectories *p* and *q* are said to be *similar* if their projections onto their movement space are close to each other throughout their lifetimes. This implies that the shape of the trajectories has no impact on the result set; so two trajectories with the same shape but that are very far away from each other will be more dissimilar than two trajectories that are very close to each other, but with wildly different shapes.

This informal notion of trajectory similarity is illustrated in Figure 4, which presents

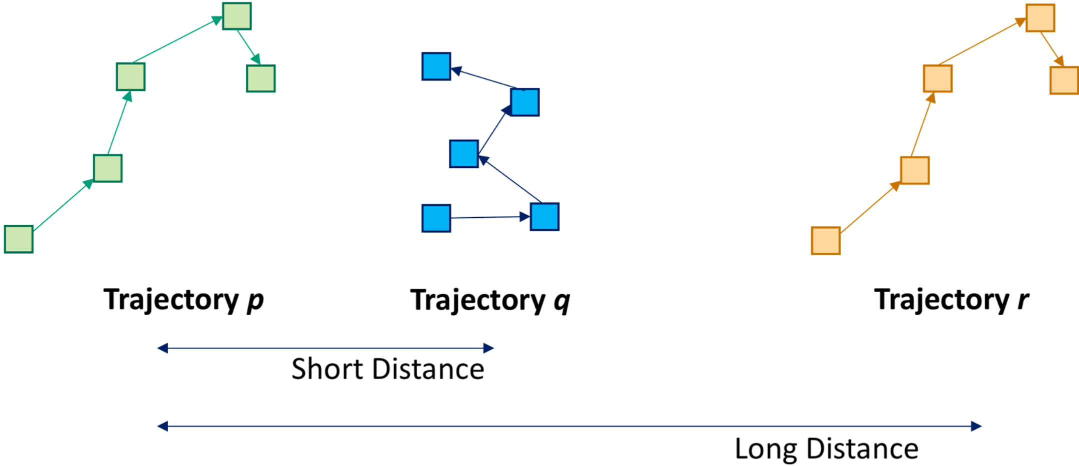

Figure 4. Example of trajectory similarity

three trajectories p, q, and r. Trajectories p and r have exactly the same shape, but are very far apart, while trajectories p and q have very different shapes, but are close to each other. Therefore, since trajectories p and q are closer to each other, the most similar trajectory to p is trajectory q.

2.3.2 Definition (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 $qother \in Q - R_p$ it is the case that $\sigma(q_{other}, p) \leq \sigma(q, p)$ [DTS08]. Figure 5 contains an example of a top-K trajectory similarity query, where the query set P consists of trajectories p_1 and p_2, and the database consists of trajectories q_3, q_4 and q_5, and $K=2$. As can be seen from this figure, the most similar database trajectories to p_1 are trajectories q_3, and q_5 because they are the closest to p_1. Similarly, the most similar database trajectories to p_2 are trajectories q_4, and q_5. For this reason, we say that the result set of this top-2 trajectory similarity query is $\{(p_1, q_3), (p_1, q_5), \{(p_2, q_4), (p_2, q_5)\}$. Notice that the actual shapes of the trajectories have no impact on the result set, just their relative proximity.

2.3.3 Applications of top-K trajectory similarity queries: Top-K trajectory similarity queries have many applications. We now discuss several of them.

- Ecology: Ecologists are interested in understanding how diseases are transmitted among birds, and how bird species make use of resources like space [VGK02] [HGKL07][CBPB10][RDTD+15]. Top-K trajectory similarity queries can help

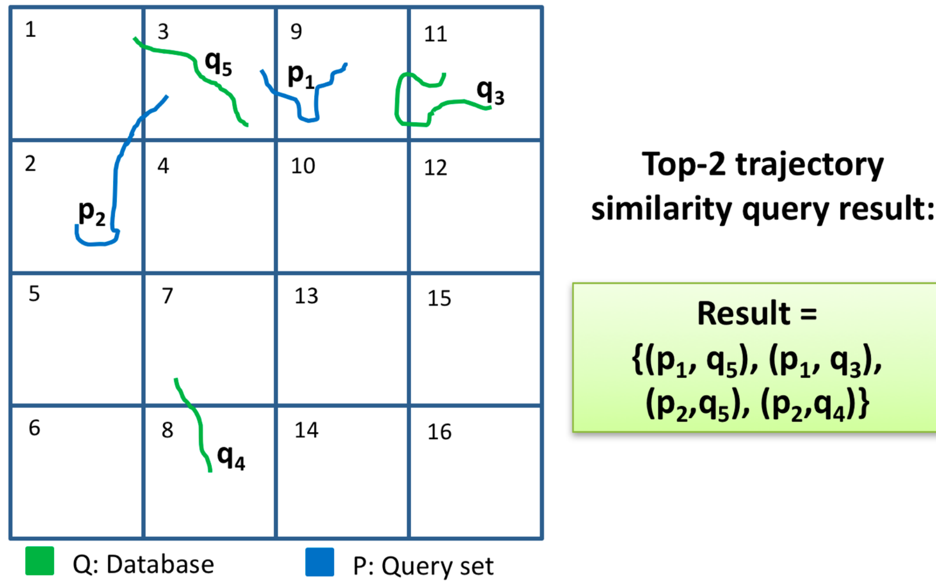

Figure 5. Example of top-K trajectory similarity query

these applications because they can help to find the birds with the *K* most similar trajectories, and this indicates birds that may come into contact with each other.

- Social Networking: These queries also have applications in online social networking sites [ZXM10] that allow sharing of travel trajectories [ML13]. For example, an individual might want to meet other people with the most similar travel trajectories to his own trajectories.
- Bioimaging: Biologists are interested in detecting spatio-temporal patterns in particle migrations during cellular mitosis [VS98][VHK06]. In particular, they are interested in finding patterns like "Type A of particles tend to seek or avoid type B particles." A top-K trajectory similarity query can help in the process of finding these patterns because if type B particles avoid type A particles, then the trajectories of both particles will be more dissimilar, thus less likely to be among each other's' top-K most similar trajectories.

- Meteorology: Meteorologists want to be able to predict the path of a developing hurricane. Since hurricanes have the tendency to take similar paths, meteorologists can use past hurricane trajectories that are similar to the one currently developing in order to help improve their predictions of its future track [BDS14][PKKF+11].
- Sports: Coaches and sports researchers are interested in knowing the movement patterns of players [COO05][BDS14] obtained from video footage of GPS sensors. For example, they are interested in deducing the common plays performed by a given team.

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 dissertation, we refer to these applications as *Big Trajectory Data applications*.

2.4 *GPU Background*

In this section we introduce GPUs, explain why and when to use them, and describe their programming model.

2.4.1 What are GPUs?

Graphics Processing Units (GPUs) are co-processors in charge of carrying out the necessary calculations to render graphical models, i.e., they are the graphics cards that are installed in desktop computers, workstations, mobile devices, etc., for displaying

graphics in a computer. As such, almost any computing device is equipped with a GPU. In most cases, GPUs are separate cards directly connected to the Peripheral Component Interconnect express (PCI express) bus, but they can also be integrated into the CPU chip or the motherboard itself.

While performing this job of rendering graphics, GPUs are required to execute the same piece of code (called *shader*) over millions of vertices under tight time-constraints [GK10]. For this reason, GPUs were designed as a parallel architecture capable of simultaneously performing many floating operations. However, nowadays, GPUs not only are designed for rendering graphics, but also can be used for general purpose parallel programming.

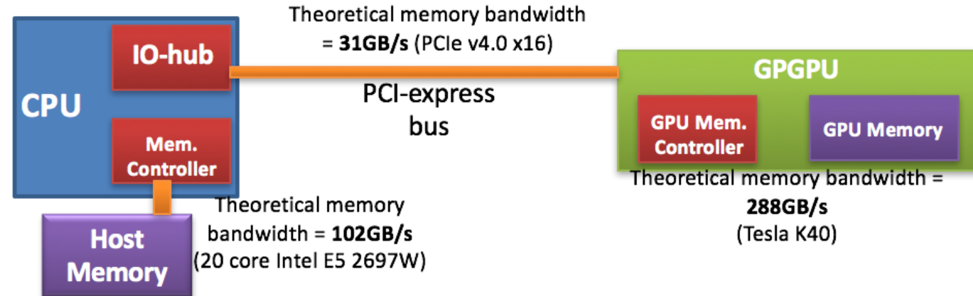

Figure 6. GPU description

2.4.2 Why and when to use GPUs?

Among the many advantages of GPUs 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 [LKCD+10]; and they are very energy

efficient [LM13]. Another advantage of GPUs is that there are works [LLZC15] that allow GPU processing from within the popular Spark parallel computing framework [ZCFS+10], so that the high instruction throughput of GPUs can be combined with the scalability, ease of use and fault-tolerance of the Spark framework. All these advantages of GPUs make them excellent tools for tackling the computational challenges associated with processing top-K trajectory similarity queries.

2.4.3 GPU Programming Model

We now discuss the programming model of GPUs [GK10] using the vocabulary of CUDA [W13], which is one of the GPU programming models. GPUs follow a parallelism model that is very similar to SIMD (Single Instruction Multiple Data) [HP12], where different threads perform the same instruction in parallel over different data. To accomplish this, the programmer must specify the total number of threads that will run in the GPU. Once this is done, during runtime, the system will assign a unique identification number to every thread; it is in this manner that different threads can perform the same instruction and work on different data, similar to what MPI (Message Passing Interface) does [P11]. GPUs are designed to run portions of code called kernels, which look like regular C-language functions and are called from within the CPU execution flow. However, there is one inconvenience with GPUs, which is that these cards have a separate memory address space from the host computer's main memory. So before kernels are launched, the CPU must call a special function to transfer the data from the host computer's main memory to the device's memory space. In a similar fashion, once the kernel finishes its execution, the CPU must call another special

function to transfer the results from the device's memory space back to the host's main memory.

GPUs can be thought of as a highly parallel architecture where execution threads form the most essential part of the execution hierarchy. At the top of this hierarchy is the *grid*, which is composed of all threads launched with the kernel. All the threads in the grid can access the GPU's global memory, which is a memory space that is big (in the order of gigabytes) and has high latency. All the threads in a grid are grouped at the time that the kernel is launched into *thread blocks*, each of which is a collection of threads that can communicate through shared memory. This is illustrated in Figure 7 which shows three thread blocks with three threads each (each GPU thread block has a number of threads which is a multiple of 32), and also shows the shared memory corresponding to every thread block. *Shared memory* is a memory space private to each thread block that is both smaller (in the order of tens of kilobytes) and faster (around 10 times) than global memory [W13]. The threads within a thread block are grouped into sets of 32 threads called warps, each of which is a collection of threads that execute the same instruction (maybe with different operands) in lockstep.

This hierarchy determines not only which threads can communicate, but also how threads can synchronize. Only the threads within a block can use barrier synchronization, and the only way to run barrier synchronization among threads belonging to different blocks is to exit the current kernel and launch a new one. The reason for this is that not all thread blocks run simultaneously.

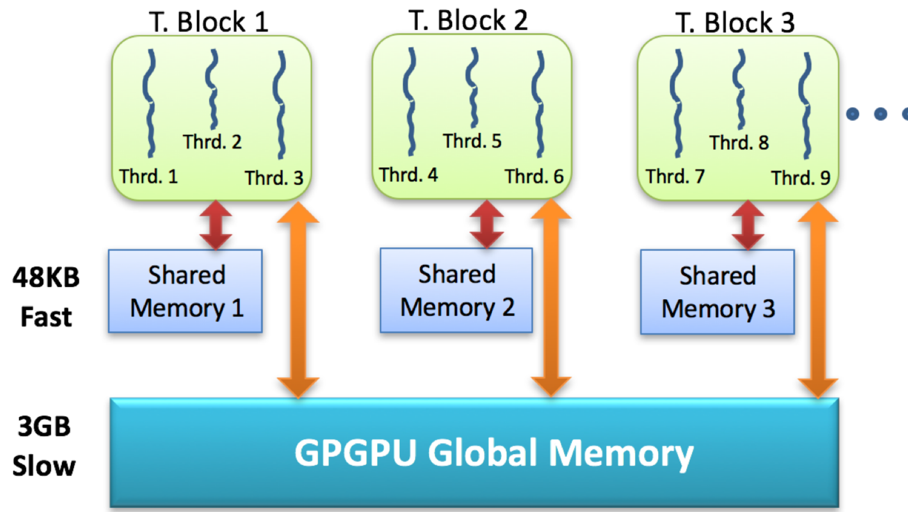

Figure 7. GPU memory space

2.4.4 GPU Issues

There are a number of challenges that need to be addressed when designing a scalable algorithm for GPUs. Among them are the five major issues: low global memory bandwidth relative to the number of threads, low PCI-express memory bandwidth, efficient use of shared memory banks, thread divergence, and load balancing. We now discuss each of these issues.

2.4.4.1 Low global memory bandwidth relative to the number of threads

In GPUs there are often thousands of threads contending for access to the (slow) global memory. This implies that every time there is a global memory read instruction, thousands of memory transactions need to be performed (one per thread). To deal with this problem, GPUs (just like regular CPUs) are equipped with caches that can exploit the spatial locality of global memory accesses in order to reduce the traffic through the

memory controller. However, in order to take advantage of such caches, threads in a GPU need to access global memory following patterns that respect the spatial locality. When threads access global memory respecting this spatial locality, then the cache can reduce the contention for memory bandwidth, in which case it is said that the GPU has *coalesced global memory accesses*.

More precisely, a coalesced memory access occurs when threads in a warp simultaneously access adjacent locations in the GPU's global memory (for example, threads $t_{32}, t_{33}, \ldots, t_{63}$ access locations $a, a + 1, \ldots, a + 31$, assuming that location a is a multiple of the size of the type located at those addresses), only a single global memory transaction is performed to access all the locations (instead of 32 separate transactions); therefore, all those potentially separate accesses to memory are coalesced into a single one. This has the advantage of reducing the demand for memory bandwidth, which in the case of GPUs constitutes one of the dominating factors for performance [KH13]. Hence, ideally, all global memory accesses within a warp should be to adjacent locations.

Figure 8 illustrates the idea of memory coalescing. In the left part of the figure we see a group of coalesced global memory accesses are coalesced because the first warp (threads 0 to 31) accesses a continuous block of memory starting at address n, which is aligned at 128 bytes. In the right part of this figure, the global memory accesses are uncoalesced because thread 31 causes the warp to accesses memory locations across two separate cache blocks.

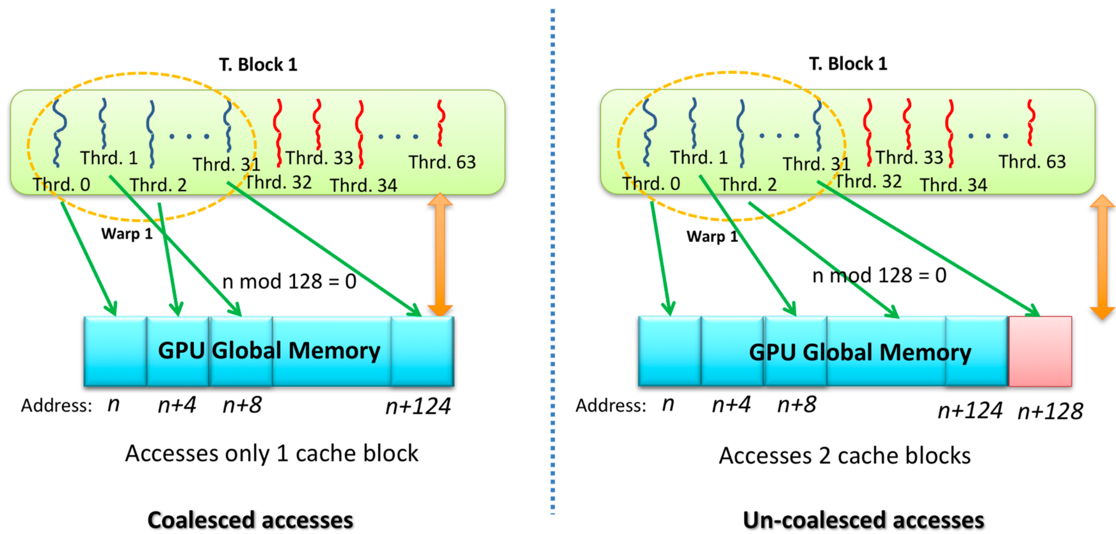

Figure 8. GPU memory coalescing

2.4.4.2 Low PCIe memory bandwidth

The GPU is connected to the host computer through the PCI express (PCIe) bus, which has a theoretical bandwidth of 31GB/s (using PCIe v4.0 x16). On the other hand, the GPU's global memory has a theoretical bandwidth of 480 GB/s (in a Tesla K80 GPU [Nvidia17]), while the host computer's main memory has a theoretical bandwidth of 85GB/s (with a 24 core Intel E7 8894 v4 chip) [Intel17]. The problem is then that transmitting data to and from the GPU is expensive because of the relatively low PCIe memory bandwidth. Therefore, GPUs should be programmed so as to maximize the amount of work performed on each data batch received through the PCIe bus, instead of communicating back and forth with the host's main memory. This problem is compounded with the fact that GPUs in general have a small global memory space (12GB ~ 24GB [Nvidia17]) so that when dealing with large volumes of data, this can generate large amounts of (slow) PCIe communication, reducing the performance of the GPU algorithm.

2.4.4.3 Efficient use of shared memory banks

To improve the performance in GPUs, the shared memory address space is divided into interleaved sub-blocks that can simultaneously and independently process transactions to this type of memory. Consequently, it is desirable that when threads in a warp make transactions to shared memory, they do not all access the same bank; instead, threads in a warp should each try to access a different shared memory bank, and in this way all these accesses can be processed independently and in parallel. To accomplish this, it is important to be aware that it is usually the case, and GPUs are not an exception in this regard, that the addresses corresponding to different banks are interleaved in a way such that, if there are 32 banks, the address x in the shared memory corresponds to the bank ($x \bmod 32$). Knowing this, it is possible to strive to equally distribute shared memory accesses among all banks.

2.4.4.4 Thread divergence

A warp is said to have *thread divergence* during execution if when a warp finds a conditional (or an iterative) statement, there is at least one pair of threads t_x, t_y such that the boolean condition of the conditional (or iterative) statement is true for t_x but false for t_y; therefore, the threads take separate paths through an if statement: one takes the if branch, and the other takes the else branch. This situation is illustrated in Figure 9, which shows a conditional statement (in the left part of the figure) that forces threads with even indexes to execute just *code A*, while the threads with odd indexes are forced to execute just *code B*. The problem with thread divergence is that it entails a

performance penalty because, since warps execute in lock-step, the hardware needs to make all threads in the warp run both branches in a serial fashion [HP12]. Ideally, code in a kernel should have no thread divergence.

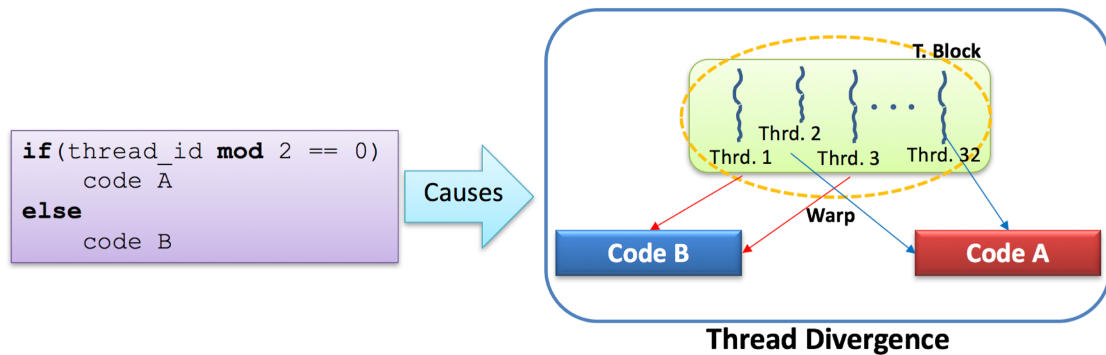

Figure 9. Thread divergence

2.4.4.5 Load balancing

Load balancing refers to striving to evenly divide the computational tasks among computing units in a way such that each GPU thread performs a similar amount of work. This issue impacts the performance of any parallel GPU algorithm because the time spent by the computational unit that receives the most time-consuming subtask will dominate the algorithm's overall execution time. Ideally all tasks submitted to the GPU should be evenly balanced among the computational units in order to ensure that no processor in the GPU is idle while others are working and, thus, no single computing unit is responsible for dominating the execution of the parallel algorithm.

3 General Issues of Top-K Trajectory Similarity Query Processing Techniques

In this section we discuss issues that should be addressed by top-K trajectory similarity query processing techniques. These issues are the different trajectory sizes in the database, local time shifts, measurement uncertainty, model uncertainty, triangular inequality, inter-trajectory sampling rate variation, intra-trajectory sampling rate variation, sampling phase variation, and the size of the parameter space.

3.1 *Different trajectory sizes*

Some techniques for computing top-K trajectory similarity queries like the Euclidean Distance (or the L^p distances in general) require that all trajectories in the dataset have the same number of points in order to be able to compute the Euclidean distance between any two trajectories [FRM94]. However, in most datasets, the trajectories have different sizes (e.g., [ZXM10]), so techniques based on L^p distances cannot compute the similarity scores between trajectories. For this reason, a top-K trajectory similarity query processing technique must use a similarity measure that can handle trajectories with different numbers of points. This situation is illustrated in Figure 10, where we see two trajectories, p with 5 points and q with 3 points. If a top-K trajectory similarity query processing technique were to use a similarity measure like the Euclidean distance, then it would not be clear which points to choose from each trajectory in order to compute the distance between them.

Requirement 1. A top-K trajectory similarity query processing technique should be able to handle databases containing trajectories with different sizes.

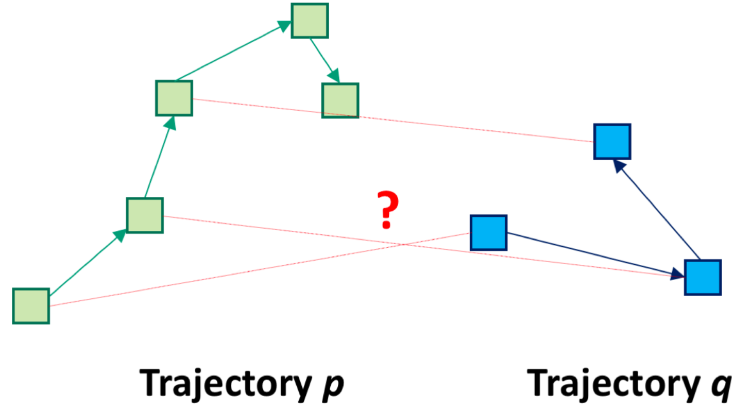

Figure 10. Different trajectory sizes

3.2 *Local time shifts*

Two trajectories *p* and *q* are said to have a *local time shift* if they have approximately the same average sampling rate, but for a portion of the lifetime of *p(q)* this *p(q)* moves faster than *q(p)*, and for the remaining portion of the lifetime of *p(q)*, *q(p)* moves faster than *p(q)*. The problem with local time shifts is that since both trajectories have the same sampling rate, then the average time elapsed between consecutive points in each of the trajectories is the same. Then, since for the initial portion of the lifetime of *p(q)* this trajectory moves faster, that means that during this initial portion, the distance between consecutive points of *p(q)* is larger than the distance between consecutive points of *q(p)* in the same interval. Figure 11 shows an example of local time shifts. To the left of the figure, we see two trajectories *p* and *q* with the same sampling rate but moving at different speeds. Trajectory *p* initially moves fast, and then moves slowly. Trajectory *q*, on the other hand, moves slowly at the beginning, and then moves fast.

The issue in this case is that if we choose to compare points in p with their corresponding points in q (respecting the order of the points in each trajectory), then the distances between the corresponding points become artificially big as seen in the left part of the figure, because the connecting segments between the corresponding points are "slanted." However, if a top-K trajectory similarity query processing technique is able to interpolate trajectories (interpolated points are shown as small circles over the trajectories on the right part of the figure), then it is possible to compute the "non-slanted" distances that better capture the true similarity between trajectories p and q.

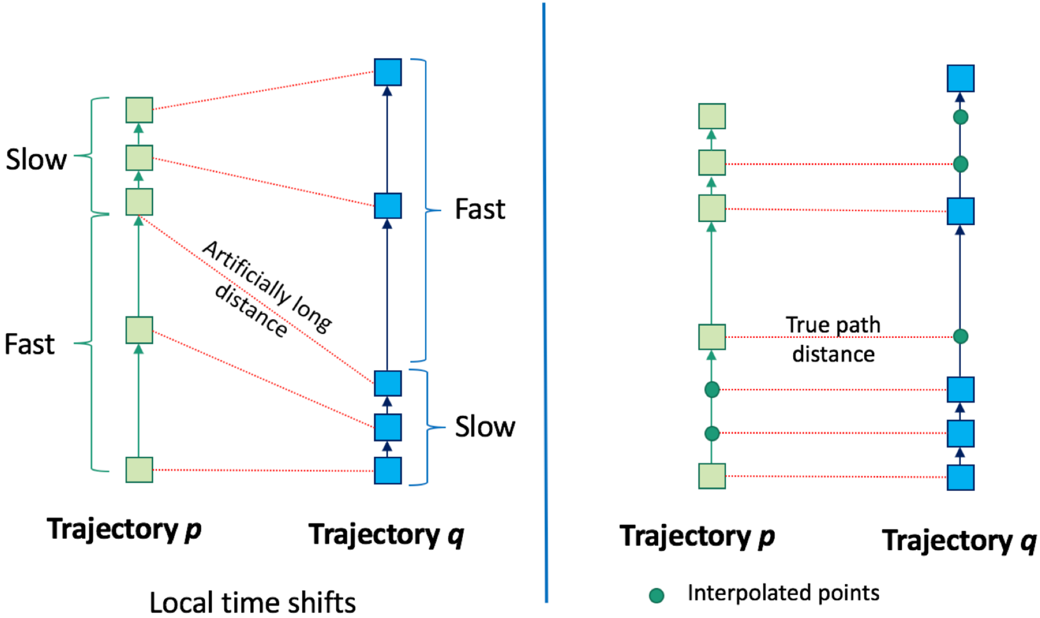

Figure 11. Local time shifts

We see then that techniques that match a point in one trajectory to at most one point in the other trajectory are not suitable for finding the similarity between trajectories with local time shifts. In practice, it is not possible to enforce that raw trajectories (i.e., trajectories without pre-processing) do not have local time shifts because the presence or absence of local time shifts depends on the relative speeds of the objects. For this

reason, a technique for top-K trajectory similarity query processing must address the issue of local time shifts.

Requirement 2. A top-K trajectory similarity query processing technique should be able to handle databases containing trajectories that have local time shifts.

3.3 *Measurement uncertainty*

Trajectory uncertainty can come from different sources that can be classified into two major classes: the sources related to the measurement / instrumentation process, and the ones related to the model dynamics. One of the noise sources related to the measurement process is the noise inherent to GPS device measurements [CBPB10][MLSC13]. This noise arises because no measurement is perfectly accurate, but also arises from the environmental conditions surrounding the sensor at the moment when the measurement is made. For example, in the ecology application (see Section I.2.3), the GPS measurement errors can be greater if there are overcast skies in the place where the animals are, or if the animals have tampered with their GPS collars, etc. This type of noise is illustrated in Figure 12, where the trajectories of three objects, *q*, *r* and *s,* are captured. Here we see that around each position (trajectory point) sampled, q_i, r_i, and s_i, for each of the three corresponding trajectories, *q*, *r* and *s*, there is an "area of uncertainty." If we ignore the measurement uncertainty, the most similar trajectory to *q* is trajectory *r*. However, if we consider the measurement to be noisy, then there is a high probability that trajectory *s* is the most similar trajectory to *q* because the points of *s* have low uncertainty and are almost as close to *q* as the points in *r.* On the other hand, the

points on *r* have large uncertainty, so there is a non-zero probability that the points r_0 and r_2 are farther away from *q* than s_0 and s_2, respectively. Hence, since many trajectory datasets are collected through the use of sensors, there is an inherent error associated to their measurements, and therefore, a top-K trajectory similarity query processing technique should be able to address measurement uncertainty.

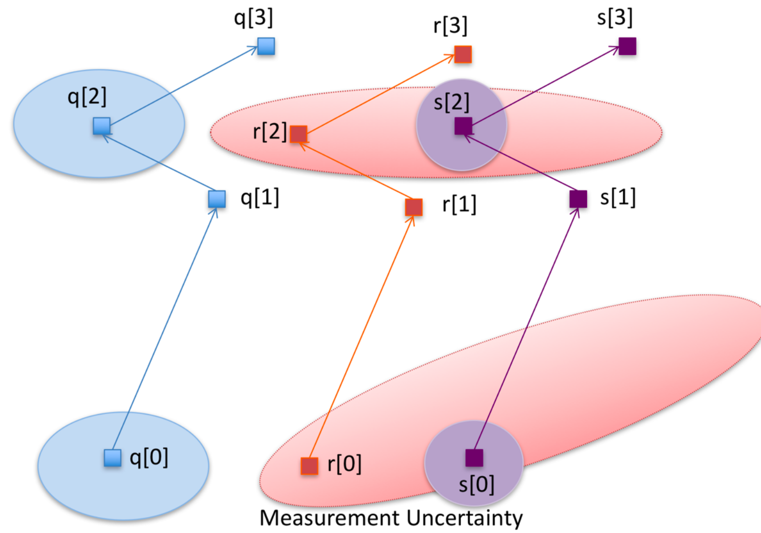

Figure 12. Measurement/instrumentation uncertainty

Requirement 3. A top-K trajectory similarity query processing technique should be able to extract the signal from the noise associated with the location measurements, and to estimate the degree of uncertainty associated.

3.4 *Model uncertainty*

One of the sources of uncertainty related to the model dynamics is the interaction between the linear interpolation model for trajectories and the inconsistencies in the sampling rate. If the time interval between two consecutive sampled points in a

trajectory is very long, i.e., the trajectory has a low sampling-rate, and the object moves at a high speed with a non-constant velocity and/or acceleration during that time interval, then the linear interpolation model underlying trajectories may not be a good approximation to the object's movement. Figure 13 shows an example illustrating this situation. The dotted lines correspond to the actual path taken by the moving objects, while the straight lines connect consecutive sampled points. If we ignore model uncertainty, then trajectory *r* is the most similar to trajectory *q* because its points are all closer to *q* than the points in *s*. However, if we consider the actual true paths, we see that *s* is the object with the most similar path to *q*. The uncertainty arises because the sampling rate was too low. This source of uncertainty is present in our animal ecology example because scientists want to maximize the lifetime of the expensive telemetry devices that they attach to animals, which makes these devices subject to energy utilization constraints. Therefore, to save energy, geolocators cannot work continuously,

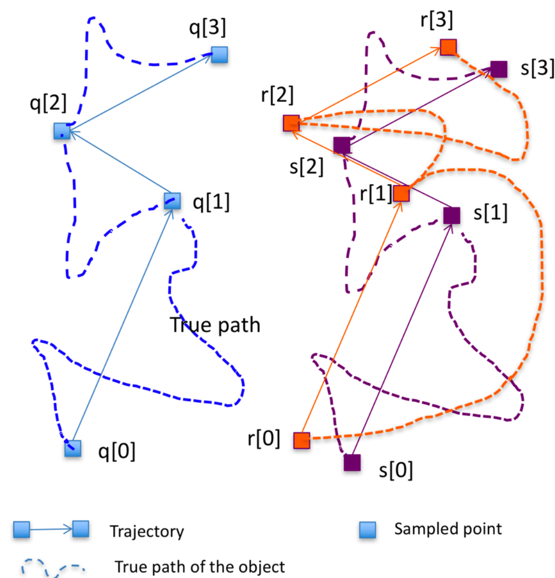

Figure 13. Model uncertainty

so animal trajectories cannot have high sampling rates [CBPB10]. As a consequence of this, there is great uncertainty in the trajectory that an animal takes between the sampled points. Another source of uncertainty in the animal ecology application relates to the fact that frequently there are missing data in the trajectories because of failed attempts at geolocation, in which the GPS device cannot successfully determine the animal's position [CBPB10]. Hence, we see that there is a concern about the impact of trajectory model uncertainty in their studies.

Requirement 4. A top-K trajectory similarity query processing technique should be able to estimate the true path of the object's trajectory when trajectories have low sampling rates.

3.5 *Triangular inequality*

Some top-K trajectory similarity query processing techniques [COO05][DTS08] exploit the fact that the underlying trajectory similarity measure satisfies the triangular inequality, in order to reduce the amount of work devoted to process the query. However, if the query processing technique does not use a similarity measure that satisfies the triangular inequality, then many existing techniques like R-trees, for which there already are parallel algorithms that work on GPUs [ZYG13], cannot be easily modified to be used with spatial data structures. This in turn entails that these latter similarity query processing techniques require ad-hoc algorithms and data structures that are far less studied than data structures like R-trees [Gutt84][BKSS90], TB-trees [PJT00] and their corresponding query processing algorithms.

Requirement 5. A top-K trajectory similarity query processing technique should ideally be able to use a similarity measure that satisfies the triangular inequality so that the knowledge of existing spatio-temporal data structures can be leveraged in order to reduce the amount of work needed to process the query.

3.6 *Inter-trajectory sampling rate variation*

This issue refers to the case when two different trajectories (whose similarity is to be computed) have very different sampling rates. In taxi trajectory datasets, cab drivers modify the GPS sensors in their taxis to reduce energy consumption [WZP12][RDTD+15]. Cab drivers that leave their GPS sensors in their default configurations have associated trajectories with a higher sampling rate than the trajectories corresponding to cab drivers that alter their GPS sensors. An example illustrating the scope of this issue when processing top-K trajectory similarity queries is the following. Suppose we are given a trajectory p of length n with a sampling rate of 15 seconds, and a trajectory q of length m with a sampling rate of 5 seconds, both moving at the same velocity. Also assume that the true paths corresponding to p and q have exactly the same shape, except that q is displaced by a fixed constant vector. We say that p and q have *inter-trajectory sampling rate variation* if they have different sampling rates. Now, when computing the similarity between p and q, every point p_i in p with timestamp t_i is likely to be matched to points q_{j-1}, q_j, q_{j+1} with timestamps t_i-5, t_i, and t_i+5, respectively (because the true paths of one of the trajectories is a translation, in a geometric sense, of the other). However, the matching between p_i and q_{j-1}, and

between p_i and q_{j+1} are both "slanted", unlike the matching between p_i and q_j. Therefore, these matches suggest that the distance (similarity) between p and q is larger than what it truly is. This is because the trajectory p is missing points with timestamps t_i-5 and t_i+5, which, if matched with q_{j-1}, q_{j+1}, respectively, would produce "horizontal" matches that better reflect the distance or similarity between p and q.

In the left-hand part of Figure 14**,** there is an example of inter-trajectory sampling rate variation. In this figure we see trajectories p and q that move at the same velocity, but trajectory p has a low sampling rate, while trajectory q has a high sampling rate. If a top-K trajectory similarity query is only allowed to compute the distances between pairs of existing points in both trajectories, then some points in p are forced to be matched to more than one point in q, and this can lead to "slanted" connecting segments that introduce artificially long distances in the computation of the similarity between p and

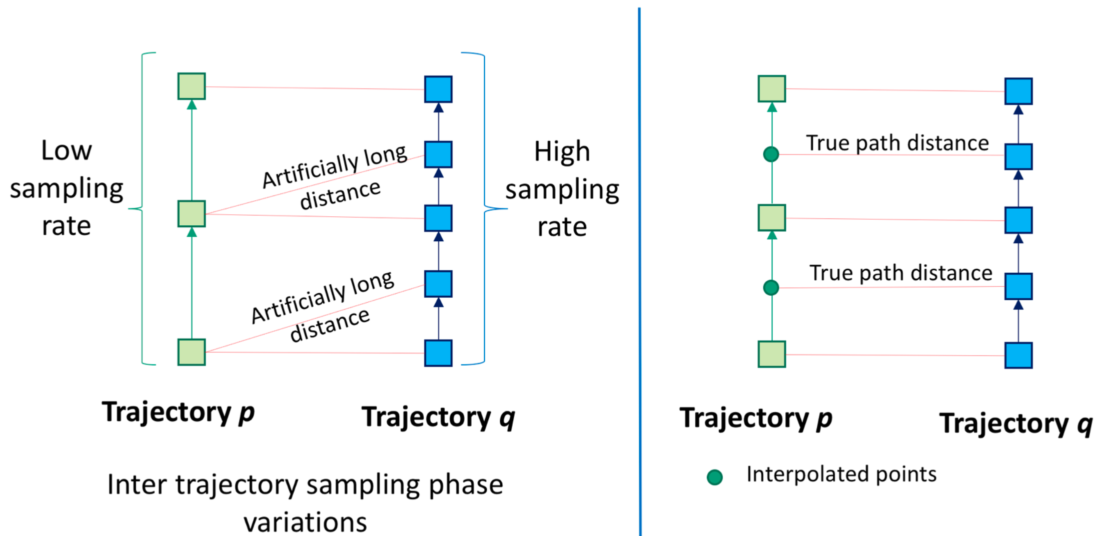

Figure 14. Inter-trajectory sampling rate variation

q. On the contrary, if a trajectory similarity query processing technique is allowed to interpolate trajectories, then it can compute distances between points in *p* and in *q* that are connected by "non-slanted" segments, leading to a more accurate trajectory similarity computation, as shown in the right-hand part of Figure 14.

Since in practical applications it is hard to enforce that all trajectories have the same sampling rate (because GPS sensors may fail, or may be running out of power), then a technique for top-K trajectory similarity query should deal with this issue.

Requirement 6. A top-K trajectory similarity query processing technique should be able to handle trajectories with different sampling rates.

3.7 *Intra-trajectory sampling rate variation*

This issue refers to the case when the sampling rate changes within a given trajectory. For example, given trajectory *p* of length *n*, if the time elapsed between points p_0 and p_1 is 5 seconds, and then between p_1 and p_2 is also 5 seconds, but then between p_2 and p_3 is 10 seconds, we say that since the time elapsed between consecutive points has changed within the same trajectory *p*, then *p* has *intra-trajectory sampling rate variation*.

Intra-trajectory sampling rate variation is an issue when processing top-K trajectory similarity queries because if the similarity between two trajectories *p* and *q* needs to be computed, and trajectory *p* and trajectory *q* both have intra-trajectory sampling rate variation, then it may be the case that initially trajectory *p* and trajectory *q* are very

similar, but after a certain point, they significantly diverge and the sampling rate becomes really low. Figure 15 illustrates the situation. In this figure we see that initially both trajectories have a low sampling rate, but then the sampling rate increases. Now, since the sampling rate is very low when *p* and *q* diverge, then there are very few points in both *p* and *q* that indicate this divergent behavior of the trajectories. Therefore, in the overall computation of the similarity of *p* and *q,* the divergent behavior section has few representing points, so the similarity measure will be biased towards those sections of *p* and *q* that have more points (where both are very similar) indicating that the trajectories are very similar, despite the fact that they strongly diverge after a certain point (which is an indication that they are not similar because, informally, for trajectories to be similar

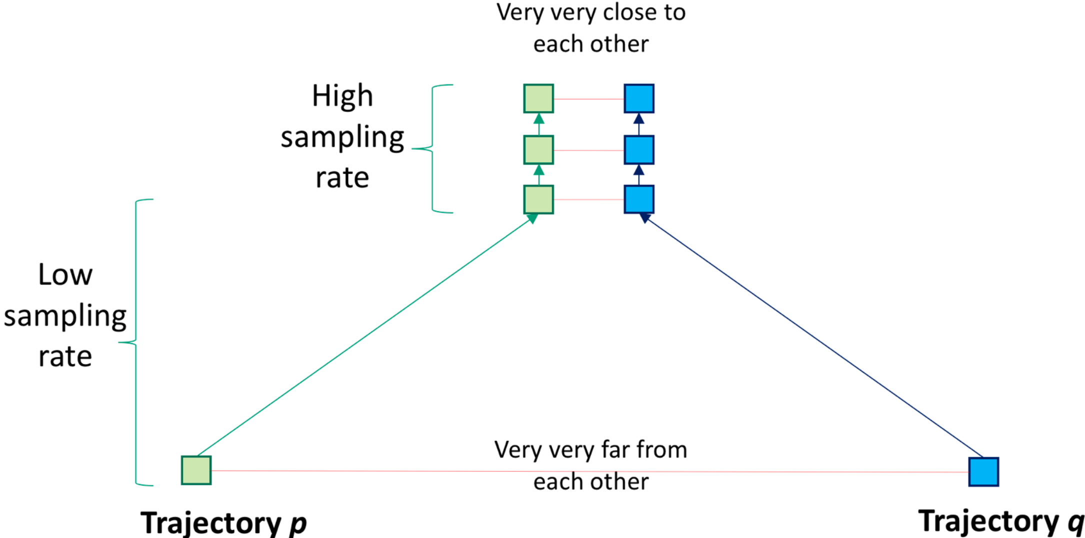

Figure 15. Intra-trajectory sampling rate variation

they must be close to each other through most of their lifetimes).

Requirement 7. A top-K trajectory similarity query processing technique should be able to handle the pairs of trajectories that have different sampling rates from each other.

3.8 *Sampling phase variation*

Assume we are given two different trajectories, p and q. They are said to have *sampling phase variation* if their corresponding true paths are approximately similar, but one is a translation (in a geometric sense) of the other. In other words, p and q have sampling phase variation if the underlying true paths satisfy that $truepath(p)[t] = truepath(q)[t+\Delta]$ for $\Delta>0$ (the phase) and any $t>0$. The left hand side of Figure 16 presents an illustration of this situation, where we see two trajectories p and q that are identical, except that one is a displacement of the other through a "rigid movement." The issue is that the true paths could be almost exactly the same, but since the true paths are sampled in an "out of phase" fashion, then this forces point $p_t = truepath(p)[t]$ to be matched with point $q_t = truepath(q)[t+\Delta]$, which could be very far away, but still on the

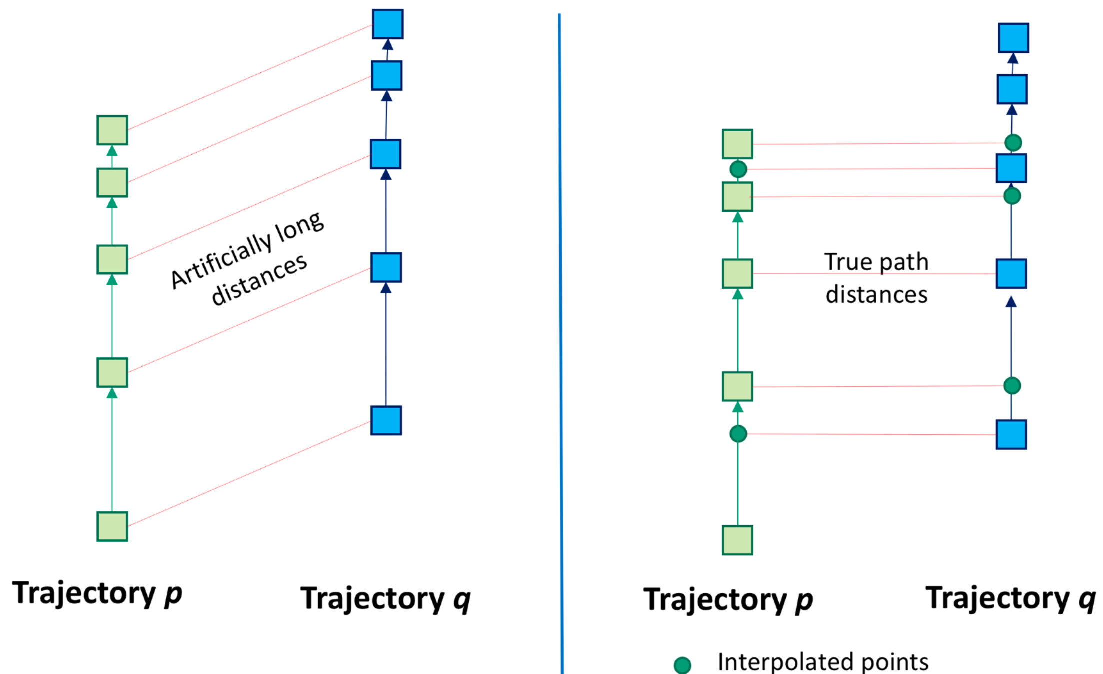

Figure 16. Sampling phase variation

sample true path. We see this in Figure 16, where the distances between the corresponding points are measured over "slanted" segments, which lead to artificially long distances. So despite the fact that the true paths are the same, since the paths are sampled differently, the matched points artificially increase the distance or similarity between the trajectories. This situation can be easily observed in datasets like the taxi trajectory dataset because it is difficult to enforce that cab drivers turn their GPS on at the same point along the trajectory path. Therefore, a top-K trajectory similarity query processing technique should address this issue.

Requirement 8. A top-K trajectory similarity query processing technique should be able to accurately compute the similarity between trajectories that share a very similar true path, but that are sampled out of phase.

3.9 *Dimensionality of the manually-tuned parameter space*

The space of parameters is the set of all possible values of all the parameters that govern the behavior of the top-K trajectory similarity query processing technique. For example, the EDR technique [COO05] depends, among other parameters, of the value of ε, which is a positive real number used by EDR to determine if two points, each belonging to a separate trajectory, are neighbors. In general, the parameters pose a difficulty for the implementation of query processing techniques in a database system because the database administrator may need to periodically and manually tune those parameters to ensure the best performance. This tuning job becomes increasingly difficult as the size of the space parameters increases and even more so if the parameters may influence one

another. Hence, a top-k trajectory similarity query processing technique should address the issue of the dimensionality of the space of parameters that need to be tuned manually by striving to reduce its size.

Requirement 9. A top-K trajectory similarity query processing technique should ideally have a space of manual parameters of low dimensionality.

3.10 *Large Databases and Large Trajectory Sizes*

In the worst case, processing top-K trajectory similarity queries can require having to compute the similarity between every query trajectory and every trajectory in the database. This problem is aggravated by the fact that in many applications (e.g., the social networking applications described in Section 2.3) there are large databases involved, which leads to big computational challenges. In addition to this, it is often the case that the trajectories themselves can have large sizes (i.e., trajectories can have many points), so that computing the similarity between trajectories is very expensive. In other words, computing a top-K trajectory similarity query is expensive because the databases are large, and because the trajectories themselves are big. One way of tackling this size issue is through the use of parallel computing architectures like GPUs, multicore CPUs, etc. However, to efficiently run algorithms on parallel architectures requires algorithms to be specifically designed to exploit their idiosyncrasies and issues (see Section 2.4.4). Therefore, a top-K trajectory similarity query processing technique should address the issue of large databases and trajectory sizes by being carefully designed to exploit parallel architectures.

Requirement 10. A top-K trajectory similarity query processing technique should be designed to exploit the idiosyncrasies of parallel architectures like GPUs in order to cope with the large databases and large trajectory sizes involved in processing this type of query.

4 Contribution

Top-K trajectory similarity queries are spatial queries of great significance because they have a wide variety of applications in many fields such as in the study of bird migration in ecology [LHW07], in the study of the movement of galaxies in astronomy [GC14][GC16], in helping identify friends with similar trajectories in social networks [ZXM10], in doing urban planning [YZXS13][WZX14], etc. On the other hand, due to the fact that trajectories can potentially have a large number of points, and that trajectory datasets can be very large, these queries pose significant computational challenges. Additionally, all trajectories have an uncertain nature because of the measurement and model errors associated with the location sensing devices used to collect the trajectories. So, despite their many applications, there exist very few works devoted to dealing with the Big Data component of trajectory query processing [ZYG12][GC14][GC16]. Out of these works dealing with Big Trajectory data, none of them is devoted to the study of top-K trajectory similarity queries; instead, these works address a very different type of trajectory similarity query called a near-join similarity query, which presents different challenges to those of the top-K trajectory similarity query, and has different applications to those mentioned above.

In this dissertation, we propose a system for processing top-K similarity queries on Big Trajectory Data using GPUs. The system consists of four techniques designed to accomplish several of the different tasks required for processing this type of query. The four techniques are TKSimGPU [LGZY15], Top-KaBT [LGZY16], TrajEstU [LGZ16] and TraclusGPU.

The first algorithm introduced in this work, called *TKSimGPU* [LGZY15], is a GPU technique for processing top-K trajectory similarity queries that is specifically designed to take advantage of the architecture of GPUs. This algorithm was conceived with the goal of ensuring load balance across the thread blocks, and providing efficient memory access patterns to help ensure memory coalescing.

The second algorithm, called *Top-KaBT* [LGZY16], is a parallel GPU pruning technique to reduce the number of spurious candidate trajectory pairs generated when processing top-K trajectory similarity queries for Big Trajectory Data applications on GPUs. 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 GPU'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. An advantage of Top-KaBT is that its ideas can be applied to any top-K trajectory similarity query processing algorithm that uses a similarity measure satisfying the triangular inequality, like ERP [CN04] and wDF [DTS08], to further reduce the amount of work necessary to process the query.

The third algorithm, called *TrajEstU* [LGZ16], is a technique for estimating the true path of moving objects in unconstrained spaces that takes into account both measurement and model uncertainty. TrajEstU works by splitting the lifetime of an object's trajectory into time intervals where the object's acceleration is nearly constant. Then TrajEstU uses the local trajectory clusters (found in an off-line preprocessing stage, so that it is done only once per database and not run each time we want to obtain a true path estimate) to obtain the movement patterns that are prevalent in the areas where trajectories have low-sampling rates, and linear regression to fit a kinematic constant acceleration model to the observed positions of the moving object. By using a linear regression model, TrajEstU reduces the uncertainty arising from the GPS measurements and the low-sampling rate of trajectories.

The fourth algorithm, called *TraclusGPU*, is a technique for performing local trajectory clustering on a hybrid GPU and multicore CPU architecture. This technique is based on the serial Traclus Algorithm [LLHW07]. It was designed because in our experiments we observed that TrajEstU's off-line preprocessing stage, which consists of finding these trajectory clusters, did not scale well with the size of the dataset. In fact, it took

weeks to run this serial algorithm on datasets of the size used in our experiments in Section IV.2.3. So, in order to make Traclus practical for Big Trajectory data, it is essential that all of its stages scale well with the size of the dataset, including the off-line preprocessing stage. The idea behind TraclusGPU consists in arranging the trajectory data in contiguous arrays, so as to allow for efficient memory accesses.

To the best of our knowledge, there does not exist a parallel GPU technique for addressing top-K trajectory similarity queries, nor a general GPU pruning technique for top-K trajectory similarity queries, nor a system that integrates uncertainty estimation, query pruning and processing on GPUs, nor a parallel GPU technique for local trajectory clustering. Our proposed techniques fill these gaps.

For performance studies, we provide analyses of the worst-case time and spatial complexities of the proposed techniques, and present extensive experimental studies of their performance in comparison with state of the art techniques. In almost all cases, our four algorithms outperform existing techniques.

5 Organization

The remaining of this dissertation is organized as follows: Chapter II presents a literature review of top-K trajectory similarity query processing techniques, and of techniques to estimate uncertain trajectories. Chapter III presents our proposed system and implementation. Chapter IV contains the theoretical and experimental analyses of our proposed approaches. Finally, Chapter V provides conclusions and future research

directions.

CHAPTER II
LITERATURE REVIEW

In this chapter we present in Section 1 a survey of some of the top-K trajectory similarity query processing techniques, and in Section 2 a survey of some of the techniques for estimating uncertain trajectories.

1 Literature Review of Top-K Trajectory Similarity Query Processing Techniques

In this section we present a survey of some of the top-K trajectory similarity query processing techniques. This survey is organized around the issues identified in Section 3 of Chapter I. Using these issues, we classified the surveyed techniques into three categories: geometry-based techniques, edit distance-based techniques and probabilistic-based techniques. The existing techniques are then classified according to the manner that they address/not address the issues and challenges identified.

1.1 *Geometry-based techniques*

Geometry-based techniques use similarity measures that are inspired on geometrical considerations. In many cases these geometry-based similarity measures satisfy the triangle inequality, which allows for efficient pruning of many results that for sure cannot form part of the query result set.

1.1.1 Euclidean Distance Technique

The simplest trajectory similarity query processing technique is the Euclidean distance top-K trajectory similarity query processing technique [FRM94]. This technique assumes that the input trajectories have the same number of points, and then computes the distance between each point in p and its corresponding point in q. After computing all these distances, it adds them all up.

The Euclidean Distance between two trajectories p and q of length n is defined as

$$L_2(p,q) = \sqrt{\sum_{k=1}^{n}(p_k.x - q_k.x)^2 + (p_k.y - q_k.y)^2}.$$

Since the Euclidean distance satisfies the triangular inequality, then this inequality can be used to help avoid an exhaustive search over the whole database, when processing top-K trajectory similarity queries. One way of doing this is by computing a lower and an upper bound for the Euclidean distance is using the triangular inequality: $d(p,q) \geq d(p,r) - d(r,q) = lowerBound(p,q)$, for a lower bound, and $d(p,q) \leq d(p,r) + d(r,q) = upperBound(p,q)$, for an upper bound.

To process top-K trajectory similarity queries, the Euclidean distance algorithm receives as inputs a query trajectory p, a database of trajectories Q, and a non-negative integer K and proceeds as follows:

1. For every trajectory p in the query set, iterate through the first k trajectories q in the database computing *upperBound(p,q)*, and adding those k trajectories to the candidate set.

2. For every other trajectory q in the database, compute *lowerBound(p,q)*. If $max(upperBound(p,c)) \leq lowerBound(p,q)$ for every trajectory c in the candidate set, then trajectory q is discarded. Otherwise, it is added to the candidate set of p.
3. Then, for every query trajectory *p*, compute *euclideanDistance(p,c)* for every *c* in the candidate set of *p*, and return the trajectories with the shortest *euclideanDistance*.

Advantages:

- This technique uses a metric to compute the similarities between trajectories, so the standard spatial data indexes (e.g., R-tree [Gutt84][BKSS90], k-d trees [Bentley75], etc.) can be used.
- This technique uses a metric that can be efficiently computed in worst-case time complexity linear in the number of points of the trajectories. This is particularly advantageous for Big Trajectory data because computing the similarities between trajectories is an expensive operation that is compounded by the facts that trajectories have large sizes, and that trajectory databases have large numbers of elements.

Disadvantages:

- The Euclidean distance is very susceptible to outlier points because an outlier point that is far away from its true position will significantly distort the final

result of the similarity. Therefore, this technique does not properly address the issue of measurement uncertainty.

- The Euclidean distance cannot cope with local time shifting since the matchings between trajectories are one-to-one and are done respecting the temporal order in each trajectory; therefore, the segments connecting corresponding points can be slanted, thereby artificially increasing the distance between the trajectories.
- The Euclidean distance can only be computed between trajectories that have the same number of points. Therefore, the Euclidean distance does not address the issue of trajectories having different sizes.
- The Euclidean distance technique was designed for serial architectures; therefore, to efficiently run in parallel architectures like GPUs, the technique would need substantial modifications.

1.1.2 Hausdorff Distance Technique

Given two trajectories, *p* and *q,* the hausdorff distance *hausd(p,q)* between them is defined as follows:

$$hausd(p,q) = max\left\{max_{p_i \in p} min_{q_j \in q} d(p_i, q_j), max_{q_j \in q} min_{p_i \in p} d(p_i, q_j)\right\}$$

In other words, the Hausdorff distance between trajectories is the maximum possible distance between a point in one trajectory to its nearest point in the other trajectory. Therefore, this trajectory similarity measure arises naturally from certain problems like the bus route comparison problem, in which a transportation authority wants to replace

one bus 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 [NJS11].

The *Hausdorff distance* is a commonly used trajectory similarity measure that has the advantage that it can be easily extended to mitigate the impact of noisy measurements. Nonetheless, since the Hausdorff distance considers a trajectory as a set of points (as opposed to a time-parametrized sequence of points), it cannot take into account the dynamics of the trajectories when measuring similarities.

The Hausdorff distance between any two trajectories p and q can be bounded above by the maximum of the Euclidean distances between points in p and points in q, which we call *upperBound(p,q)*, and bounded below by the minimum of such distances, which we call *lowerBound(p,q)*. To process top-K trajectory similarity queries, the Hausdorff technique receives as inputs a query trajectory p, a database of trajectories Q, and a non-negative integer K and proceeds as follows:

1. For every trajectory p in the query set, visit the first K trajectories q in the database computing *upperBound(p,q)*, and adding those K trajectories to the candidate set.
2. For every other trajectory q in the database, compute *lowerBound(p,q)*. If $max\left(upperBound(p,c)\right) \leq lowerBound(p,q)$ for every trajectory c in the candidate set, then trajectory q is discarded. Otherwise, it is added to the candidate set of p.

3. Then, for every query trajectory *p*, compute *hausd(p,c)* for every *c* in the candidate set of *p* and return the trajectories with the shortest wDF.

Advantages:

- The first advantage of the Hausdorff distance technique is that it uses the Hausdorff similarity measure, which can deal with trajectories of different sizes.
- Another advantage of the Hausdorff distance technique is that it is parameter-free, so there is no need to search in a parameter space for the right parameter values.
- A third advantage of the Hausdorff distance technique is that it is a metric, so it satisfies the triangular inequality. Hence, it can be used with well-studied spatial data indexes like R-trees, k-d trees [Bentley75], etc.

Disadvantages:

- One of the main disadvantages of the Hausdorff distance technique is that, since it uses the Hausdorff distance to compute the similarity between trajectories, it does not take the dynamics or the relative order of the points in a trajectory into account to compute its similarity with another trajectory.
- Another disadvantage of this technique is that it does not address the issues of measurement and model uncertainty. However, when compared against the Euclidean distance technique, it is more robust against outliers. This is because a single outlier point can introduce significant errors in the Euclidean distance

technique; however, in the Hausdorff technique, a single outlier may not necessarily affect the overall similarity.

1.1.3 w-constrained Discrete Fréchet Distance (wDF)

Ding, Trajcevski, and Scheuermann proposed in [DTS08] a top-K trajectory similarity query processing algorithm based on two key ideas. The first key idea is using a modification of the discrete Fréchet distance, called the w-constrained discrete Fréchet distance, as a way to efficiently measure the similarity between trajectories, which helps overcome the problem with the Hausdorff distance, which ignores the dynamics of the trajectories involved. The second key idea consists in proposing upper and lower bounds for the Fréchet distance, and using those two bounds to avoid an exhaustive search when answering top-K trajectory similarity queries. We will now comment on these two key ideas.

The first key idea introduced in the work [DTS08] is the w-constrained discrete Fréchet distance. However, since the w-constrained discrete Fréchet distance builds up on the continuous Fréchet distance and its discrete version, we first present these last two similarity metrics. Given two continuous curves in the plane $f\colon[a_1,b_1] \to \mathbb{R}^n, g\colon[a_2,b_2] \to \mathbb{R}^n$, the *Fréchet distance* between them is defined as $fd(f,g) = \inf_{\alpha,\beta} \max_{t\in[0,1]} \|f(\alpha(t)) - g(\beta(t))\|$, where α and β are continuous and monotonous functions $\alpha\colon[a_1,b_1] \to [0,1]$, $\beta\colon[a_2,b_2] \to [0,1]$ called *parametrizations*.

To intuitively explain the Fréchet distance, we can consider one of the objects to be a person, and the other object as a dog. We also assume that neither the person nor the dog backtrack or go backwards along the trajectories they describe (this is the intuition behind a monotonous parametrization), and that they can change their velocity (the parametrizations can be considered as changes in the velocity of the objects). Then, the Fréchet distance between the curves they describe is the length of the shortest leash necessary to perform the walk (this is the reason why the formal definition takes the infimum). We see then that this is how the Fréchet distance takes the dynamics of the trajectories into account.

The disadvantage of the continuous Fréchet distance, when applied to trajectories, is that trajectories are discrete objects. Second, computing the Fréchet distance between two trajectories would require solving an optimization problem over the space of all possible parametrizations α and β. To address these issues, the work [EM94] introduced a discretized version of the Fréchet distance that can be applied to trajectories. This distance, just like the continuous Fréchet distance, computes the similarities between trajectories by pairing a point in one trajectory with a point in the other. Nonetheless, these pairings ignore the temporal distances and that can lead to distortions in the similarity between the trajectories [DTS08]. To address this issue, Ding Trajcevski and Scheuermann proposed the w-constrained discrete Fréchet distance.

Given two trajectories p and q, their *w-constrained Discrete Fréchet distance* is defined as: $wDF(p,q) = \min \{|C_w| : C_w \text{ is a coupling between } p \text{ and } q \text{ s.t. if } (p_{a_i}, q_{b_i}) \in C_w \Rightarrow |p_{a_i}.t - q_{b_i}.t| < w\}$, where a w-constrained coupling between p and q is a sequence $\{(p_{a_1}, q_{b_1}), (p_{a_2}, q_{b_2}), \dots, (p_{a_k}, q_{b_k})\}$ with $p_{a_i} \in p$, and $q_{b_i} \in q$, and such that $a_1 = b_1 = 1$, $a_k = b_k = |p| = |q| = n$, and for all *i:* $a_{i+1} = a_i$ or $a_{i+1} = a_i + 1$, and $b_{i+1} = b_i$ or $b_{i+1} = b_i + 1$ (the matchings are monotonic non-decreasing), and the length of a constrained coupling is $|C_w| = \max\{d(p_{a_i}, q_{b_i})\}$.

The intuition behind trajectory couplings is that each pair in a coupling represents the state of the leash during any time window of length less than *w*. On the other hand, the length of a coupling is, by definition, the largest distance between any pair of points in the coupling. Then, it is easily seen that minimizing the length of the coupling is a discretization of the Fréchet distance because the parametrizations correspond to the couplings, and the infimum over all parametrizations corresponds to the minimum length coupling.

The second key idea proposed in the work in [DTS08] is the introduction of both a lower and an upper bound to the wDF between two trajectories, which are much cheaper to compute than the actual wDF. The lower bound is based on the idea that for any trajectory one can obtain a sequence of MBRs that contain the trajectory at disjoint time intervals. To find a lower bound to the wDF distance between two trajectories p and q, [DTS08] proposes finding a sequence of MBRs for each trajectory. A *w-constrained lower (upper) bound coupling* LBwDF [UBwDF] is a monotonous coupling

between the MBRs of p and q (instead of between the points of p and q), where the link is defined as *minDistance[or maxDistance]* between MBRs. The *w-constrained lower bound distance* between p and q is the minimum length of all possible w-constrained lower bound couplings. An advantage of this lower bound is that it can be computed using the same algorithm for computing the wDF distance, but only changing the link. A similar idea is used for the upper bounds.

To process top-K trajectory similarity queries, [DTS08] receives as inputs a query trajectory *p,* a trajectory database *db* and a non-negative integer *K*, and proceeds as follows:

1. For every trajectory *p* in the query set, iterate through the first *K* trajectories *q* in the database computing *LBwDF(p,q)*, and adding those *K* trajectories to the candidate set.
2. For every other trajectory q in the database, compute *LBwDF(p,q).* If $max\left(UBwDF(p,c)\right) \leq LBwDF(p,q)$ for every trajectory *c* in the candidate set, then trajectory *q* is discarded. Otherwise, it is added to the candidate set of *p*.
3. Then, for every query trajectory *p*, compute *wDF(p,c)* for every *c* in the candidate set of *p* and return the trajectories with the shortest wDF.

Advantages:

- The wDF top-K trajectory query processing algorithm addresses the issue of trajectories having different sizes.

- A second advantage of this query processing algorithm is that it uses the Fréchet distance as a similarity measure. The Fréchet distance is a pseudo-metric, so it satisfies the triangular inequality. Therefore, this technique can also be used with well-known spatial data structures like R-trees.

Disadvantages:

- The wDF top-K trajectory similarity query processing technique does not address the issues of local time shifting.
- A second disadvantage of the wDF technique is that it does not address the issues of measurement uncertainty and of model uncertainty.
- A third disadvantage is that the wDF top-K trajectory similarity query processing algorithm takes an input parameter w (the temporal constraint for the couplings), so it is not a parameter-free algorithm. Therefore, to use this query processing algorithm, there is a need to search the value of w that yields the best performance.
- Finally, the wDF algorithm is a serial algorithm, so it requires significant changes in order to run in parallel architectures like GPUs.

1.1.4 DISSIM

Frentzos, Gratsias and Theodoridis introduced in [FGT07] the first top-K trajectory similarity query processing algorithm, called DISSIM, that addresses the issue of inter-trajectory sampling rate variations by using linear interpolation, so that it can avoid computing the distance between points that lead to artificial increases in the distance

between trajectories. Another peculiarity of the DISSIM trajectory query processing algorithm is that it computes trajectory similarities by taking the temporal dimension into consideration, that is, two trajectories are similar if they are closer to each other in both time and space and not just in space alone.

DISSIM uses the following (dis)similarity measure between two trajectories, p and q: $DISSIM(p,q) = \sum_{k=1}^{n-1} \int_{t_k}^{t_{k+1}} D_{p,q}(t)\,dt$, where $\{t_k | 1 \leq k < n\}$ is the set of timestamps of both p and q, and $D_{p,q}(t)$ is the Euclidean distance between trajectory p and trajectory q at time t. As we have discussed, unlike many of the techniques proposed in this area, DISSIM computes the (dis)similarity between trajectories by taking the time component into consideration. This is because the DISSIM measure is a function of the time instants when the points in the trajectories were sampled. Therefore, DISSIM can tell if the true paths corresponding to two trajectories are close to each other *at a specific time interval*, unlike most of the techniques proposed in this area, which can only tell if the true paths of the trajectories are close *ignoring time*.

To avoid an exhaustive search algorithm, Frentzos, Gratsias and Theodoridis proposed in [FGT07] a lower bound, called OPTDISSIM, to the DISSIM dissimilarity that is cheaper to compute than DISSIM, and an upper bound, called PESDISSIM, to DISSIM that is also cheaper to compute than DISSIM. This lower bound is used in their algorithm to remove trajectory candidates that for sure cannot be part of the query result set. The idea of using this lower bound OPTDISSIM is that if for a trajectory not seen so far, its OPTDISSIM is greater than the DISSIM of K trajectories seen so far, then

since OPTDISSIM is a lower bound to DISSIM, the DISSIM of this unseen trajectory cannot be smaller than that of the *K* trajectories seen so far. Therefore, the unseen trajectory can be pruned away.

To process top-K trajectory similarity queries, DISSIM receives as inputs a query trajectory *p,* a trajectory database *db* and a non-negative integer *K*, and proceeds as follows:

1) Insert all the segments of all trajectories in the database into an R-tree.
2) Visit the R-tree in best-first mode using the MINDIST between the query trajectory *q* and the Nodes and Leaves as heuristic. This means that the nodes and leaves are visited in increasing order of MINDIST.
3) If the algorithm encounters a leaf node, then for every entry of that leaf node if that entry belongs to a trajectory that has been pruned away, then that entry is ignored. Otherwise, the algorithm retrieves the object *o* (trajectory) corresponding to that entry and if the temporal extent of *q* and the temporal extent of the entry intersect, the algorithm adds that intersecting time interval to a list L_O of time intervals associated with the entry's object *O*.
4) If L_O contains all intervals spanned by the temporal extent of *o*, then *o* is added to a list *Completed* whose elements are all those objects, and then the algorithm computes $d_o = DISSIM(p,o)$. If d_o is greater than the DISSIM of the *K* most similar objects to *q* seen so far, then *o* is discarded. Otherwise, it and the *K* most similar objects to *q* seen so far are recomputed. If L_o does not contain all intervals spanned by the temporal extent of *o*, the algorithm computes the

PESDISSIM between q and o and proceeds analogously. If PESDISSIM is smaller than the similarity of the K most similar objects to q seen so far, it is stored in that list of similar objects.

5) When all the entries in the R-tree have been visited, the algorithm outputs the K most similar objects to q.

Advantages:

- The DISSIM top-K trajectory similarity query processing technique does not depend on any tuning parameter so that the dimensionality of the space of manually-tuned parameters is 0, so it is easy to use this technique to process trajectory similarity queries without having to search for a good parameter value in a large parameter space.
- A second advantage of DISSIM is that it addresses the issue of inter-trajectory sampling rate variation because this technique does interpolation in the trajectory with lower sampling rate. This way, the "slanted matchings" (mentioned in Section I.3) are avoided and the computed score between trajectories better reflects the similarity between trajectories.
- Unlike many techniques to process top-K trajectory similarity queries, which use ad-hoc data structures, DISSIM uses an R-tree [Gutt84][BKSS90], a well-known technique, as the main data structure to store and retrieve the trajectories in the database. Among other things, using an R-tree has the advantage that DISSIM could potentially be implemented in parallel [ZYG13].

Disadvantages:

- DISSIM computes the (dis)similarity between two trajectories by considering one-to-one matches. This means that DISSIM cannot deal with local time shifts (like DTW, which addresses local time shifts by allowing points in one trajectory to match with *multiple* points in the other trajectory).
- Another disadvantage of DISSIM is related to the way it deals with the inter-trajectory sampling rate issue. Since DISSIM performs interpolation in the low sampling rate trajectories to find a better point to match against the other trajectory, if the sampling rate is low enough and the true path of the object is sinuous enough, then the trajectory interpolation model will severely deviate from the true path, so that the similarity score will not truly reflect how similar the two trajectories are. This problem with the trajectory interpolation model has been explained in more detail in Section I.3.
- A third disadvantage of DISSIM is that it is a serial algorithm, so it requires substantial changes for it to work efficiently on parallel architectures like GPUs.

1.2 *Edit distance-based Techniques*

Edit distance-based techniques use variations of the edit distance of strings [CLRS09] to measure the similarity between two trajectories. In general, these techniques address the issues of local time shifts and sampling phase variation, but do not satisfy the triangular inequality, so they usually require ad-hoc indexing data structures to avoid exhaustive searches when processing top-K trajectory similarity queries.

1.2.1 Dynamic Time Warping (DTW)

Dynamic Time Warping (DTW) is a top-K trajectory similarity query processing technique introduced in [BK94][KP00]. It computes the similarity between trajectories in the following way. Assume two trajectories p, of length m, and q, of length *n, are given*. The intuition behind the DTW similarity measure can be seen in terms of an optimization problem. A match *M* between p and q is a relation over the set $[1, m] \times [1, n]$ that satisfies the following properties: (i) There are no crossings, i.e., if the pair (i, j) belongs to *M*, then all other pairs in *M* of the form (k, l) with $k \geq i$ satisfy that $l \geq j$. Also, all other pairs of the form (k, l) with $l \geq j$ satisfy $k \geq i$. (ii) For every integer $i \in [1, m]$ there exists a pair of the form $(i, j) \in M$, for some $j \in [1, n]$. (iii) For every integer $j \in [1, n]$ there exists a pair of the form $(i, j) \in M$ for some $i \in [1, m]$. Every match M on p and q has an associated cost that is computed by adding up the costs of each individual pair contained in *M*. The cost of the pair $(i, j) \in M$ is given by $d(p[i], q[j])$. Therefore, the cost of a match *M* is given by the expression $\sum_{(i,j) \in M} d(p[i], q[j])$.

The optimization problem behind DTW is to find the least-cost matching on p and q. It turns out that this optimization problem can be solved with dynamic programming. Let's see why: Suppose, again, that we are given two trajectories *p*, of length *m*, and *q*, of length *n*, and that neither *m* nor *n* is zero. Also assume that $DTW(m, n)$ is the cost of the least-cost matching on p and q. Then as a consequence of the three properties mentioned above, an optimum match for p and q must contain the pair *(m,n)*. Therefore, $DTW(m, n)$ must be equal to $d(p[m], q[n]) + other\ cost$. According to the three properties explained before, there are three mutually exclusive cases for this optimum

match. Either this match contains a pair *(m-1,n)*, in which case $DTW(m,n) = d(p[m],q[n]) + DTW(m-1,n)$; or the match contains the pair *(m,n-1)*, in which case $DTW(m,n) = d(p[m],q[n]) + DTW(m,n-1)$; or the match contains neither *(m-1,n)* nor *(m,n-1)*, so that $DTW(m,n) = d(p[m],q[n]) + DTW(m-1,n-1)$. Since these three cases are exhaustive, we have that $DTW(m,n) = d(p[m],q[n]) + min\{DTW(m-1,n), DTW(m,n-1),\ DTW(m-1,n-1)\}$.

The DTW similarity measure can be computed in worst-case time complexity $O(m \times n)$, and with worst-case space complexity $O(m \times n)$ (if we want to retrieve the matching). Now, to process a top-K trajectory similarity query using the DTW similarity measure, there are pruning techniques [YJF98][KR05] based on lower and upper bounding the DTW of any two trajectories. These lower and upper bounds can be used to help process top-K trajectory similarity queries more efficiently. We now explain how this is done. DTW receives as inputs a query trajectory *p*, a database of trajectories *Q*, and a non-negative integer *K* and proceeds as follows:

1. For every trajectory p in the query set, visit the first *K* trajectories q in the database computing *upperBound(p,q)*, and adding those *K* trajectories to the candidate set.
2. For every other trajectory q in the database, compute *lowerBound(p,q)*. If $max\big(upperBound(p,c)\big) \leq lowerBound(p,q)$ for every trajectory *c* in the candidate set, then trajectory q is discarded. Otherwise, it is added to the candidate set of *p*.

3. Then, for every query trajectory *p*, compute *DTW(p,c)* for every *c* in the candidate set of *p* and return the trajectories with the shortest *DTW*.

Advantages:

- A first advantage of this top-K trajectory similarity query processing technique is that it addresses the issue of trajectories with different sizes (number of points).
- Another advantage of DTW is that, compared to Euclidean distance-based techniques, it is significantly less sensitive to outliers [WMDT+13].

Disadvantages:

- The DTW does not satisfy the triangular inequality, so it is not a metric. Therefore, standard spatial indexes like the R-tree and the TB-tree cannot be used.
- The DTW similarity measure forces all points to participate in the optimum match, even outliers.

1.2.2 Longest Common Subsequence (LCSS)

The Longest Common Subsequence (LCSS) top-K trajectory similarity query processing algorithm was proposed in [VKG02] to improve upon the Euclidean distance and the DTW similarity measures to better handle measurement noise. LCSS measures the similarity between two trajectories *p* and *q* by counting the number of points shared in common between them, where a point in *p* and a point in *q* are shared in common by

p and q if they are sufficiently close to one another. Therefore, it is seen that one immediate advantage of this measure is that not all points of both trajectories have to be matched. This similarity measure is a generalization of the Longest Common Subsequence of Strings, where the idea is to find a sequence (not necessarily made up of characters that appear consecutively in any of the strings) of maximum length that is contained in both input strings.

Before proceeding to explain how LCSS computes the similarity between two trajectories, we first define what it means when two points match with respect to a positive real number. Given a number $\epsilon > 0$, and two points p_1 and p_2, these points are said to *match* with respect to ϵ if $|p_1.x - p_2.x| < \epsilon$ and $|p_1.y - p_2.y| < \epsilon$. With this definition, we now proceed to explain the intuition behind the similarity measure. Suppose, again, that we are given $\epsilon > 0$, and two trajectories p, of length m, and q, of length n, and that neither m nor n is zero. Also assume that $LCSS(m, n, \epsilon)$ is the length of the least common subsequence of p and q for ϵ. If $p[m-1]$ matches $q[n-1]$ with respect to ϵ, then we know that the LCSS of p and q for ϵ contains $p[m-1]$ (and $q[n-1]$), so that the LCSS of p and q is equal to the LCSS of $p[m-1]$ and $q[n-1]$ and then appending $p[m-1]$ (or $q[n-1]$). Otherwise, the LCSS of p and q is equal to the $LCSS(p, q[n-2])$ or to $LCSS(p[m-2], q)$. We see then that the LCSS similarity measure between any two trajectories of lengths m and n can be computed with a dynamic programming in worst-case time complexity $O(m \times n)$, and with worst-case space complexity $O(m \times n)$, if we want to retrieve the matching sub-sequence, or $O(max(m, n))$ if we do not.

Advantages:

- One advantage of the LCSS query processing algorithm is that it addresses the issue of different trajectory sizes.
- The LCSS addresses the issue of local time shifting.
- The LCSS addresses the issue of measurement uncertainty.

Disadvantages:

- The first disadvantage of the LCSS top-K trajectory similarity query processing algorithm is that it does not satisfy the triangular inequality; therefore, standard indexes like the R-tree, k-d tree [Bentley75], etc. cannot be used with it.
- A second disadvantage of this top-K trajectory similarity query processing algorithm is that it does not address the issue of model uncertainty, so that in trajectories with low-sampling rates it may produce inaccurate query results.
- Another disadvantage of LCSS is that it is a serial algorithm with $O(m \times n)$ worst-case space and time complexity; therefore, it does not scale for Big Trajectory Data. Moreover, it requires substantial modifications in order to run efficiently on parallel architectures like GPUs and multicore CPUs.

1.2.3 Edit Distance on Real Sequence (EDR)

The Edit Distance on Real Sequence (EDR) [COO05] is a serial top-K trajectory similarity query processing algorithm designed to address the problem of local time shifts, and noise sensitivity when computing the similarity between trajectories. This technique represents an improvement upon ERP, DTW and LCSS because it is less

sensitive to outliers than these three latter techniques. It is also an improvement upon the Euclidean distance because of this same reason, and because it does not require trajectories to all have the same length.

The idea behind this similarity measure is to compute the minimum number of modifications (insertions, deletions) that need to be performed on one trajectory to transform it into the other trajectory. As such, this similarity measure is based on the edit distance of strings. The difference is that instead of comparing strings character by character (where two characters match if and only if they are the same), EDR compares trajectories point by point, where two points match if one of them is within an $\epsilon > 0$ Euclidean distance of the other.

The worst-case time complexity of computing the EDR similarity between two trajectories p and q of lengths m and n, respectively, is $O(m \times n)$. Therefore, computing a top-K trajectory similarity query on a large database is infeasible for large databases containing trajectories with many points. This is the reason why the EDR technique requires additional pruning techniques. To solve this problem, [COO05] introduced three EDR-specific pruning techniques: pruning by near triangle inequality, pruning by mean value q-gram, and pruning by histograms.

Now, we explain the rationale behind pruning by near triangle inequality. It has been proved in [COO05] that the EDR similarity satisfies the following inequality: $EDR(q,s) + EDR(s,r) + |s| \geq EDR(q,r)$, for any trajectories q, r and s. To prune

using the Near Triangle Inequality, the technique selects a subset X of trajectories in the database *db*, and computes and stores the exact EDR distance $EDR(q, x)$ for every x in X. After this, it sorts these trajectories x in X in ascending order of their EDR distance to q. Then, the algorithm iterates through every trajectory s in the *db*, and finds the following lower bound to $EDR(q, s)$, called *maxPruneDist,* using the near triangle inequality. If *maxPruneDist* (the lower bound to $EDR(q, s)$) is greater than the K-th smallest EDR distance in $EDR(q, x)$ for x in X, then we know that s cannot form part of the result set, so it is discarded without having to compute $EDR(q, s)$. Otherwise, we compute $EDR(q, s)$ and insert it into the subset X in ascending order of EDR distance to q.

Pruning by mean value-q gram and pruning histograms work in a similar way as pruning by near-triangle inequality in that they consist of obtaining lower and upper bounds (which are much cheaper to compute than the EDR similarity) for the EDR similarity between two trajectories, and then using these bounds to quickly remove candidates that cannot form part of the query result set.

Advantages:

- One advantage of EDR is that it addresses the issue of trajectories having different lengths.
- This technique also addresses the issue of local time shifts in trajectories and is more robust than the Euclidean distance and the ERP distance in terms of measurement noise.

Disadvantages:

- One disadvantage of the EDR distance is that it is not a metric; hence, regular and well-studied spatial indexes like TB-trees, R-trees, M-tree [CPZ97] cannot be used to reduce the number of spurious candidate pairs.
- One consequence of this matching between points in the trajectories is that the EDR similarity measure does not satisfy the triangular inequality. Therefore, traditional pruning techniques like TB-trees, kd-trees [Bentley75] and R-trees cannot be used to avoid computing the EDR similarity between the query trajectory and every trajectory in the database.
- Another disadvantage of EDR is that it is a serial algorithm and requires substantial modifications to efficiently run in parallel architectures like GPUs.

1.2.4 Edit Distance with Real Penalty (ERP)

The Edit distance with Real Penalty is a serial top-K trajectory similarity query processing technique proposed in [CN04] to improve upon DTW and LCSS by being more robust in circumstances where trajectories have measurement uncertainty, and to improve upon EDR by satisfying the triangular inequality.

The key idea behind the ERP top-K trajectory similarity query processing technique is, just like in the case of the EDR trajectory similarity query processing technique, borrowed from the edit distance of strings. In fact, ERP and EDR are very similar to each other, the difference between them is the distance between a point in a trajectory *p*

and another in a trajectory q is not discretized with values 1 or 0 (like in EDR), but instead their Euclidean distances are computed. Since the matchings between points are not discretized, but instead their true distances are computed, ERP can be shown to satisfy the triangular inequality.

The triangular inequality, which the ERP distance satisfies, can be used to prune candidates of top-K similarity trajectories. If q is a query trajectory, and s and r are database trajectories, then the triangular inequality states that $ERP(q,s) \geq ERP(q,r) - ERP(r,s)$. This inequality is used by ERP as follows. The algorithm first computes $ERP(r_i, s)$ for every r_i belonging to a subset R of the trajectory database, and then computes $ERP(q, r_i)$. Once this is done, the algorithm sorts these trajectories in R in increasing order of ERP distance to q. Then to process a top-K trajectory similarity query, the algorithm performs a linear scan visiting every trajectory s in the database, computing $max\big(ERP(q, r_i) - ERP(r_i, s)\big) \leq ERP(q, s)$. If the left-hand side of this inequality is greater than the K-th smallest ERP distance found so far, then s can be discarded. Otherwise, s could be in the result set of the query, so the distance $ERP(q, s)$ is computed, and stored in the set of ERP distances found.

One problem with the previous technique is that it performs a linear scan through the database. By using a spatial tree index, it could be possible to avoid having to linearly explore all database trajectories. However, such spatial tree index would need to be two-dimensional. The work by [CN04] proposed another pruning technique to avoid having to use a two-dimensional spatial structure. Instead, they proposed a new lower

bound for the ERP distance: $DLB(q,s) = \left|sum(q) - sum(s)\right| \leq ERP(q,s)$. This technique uses a B+-tree [Bayer71] to index all the trajectories in the database by their sum(s). Then, it performs a range search in the B+-tree using $sum(q)$ to retrieve the K trajectories and compute their ERP distances to q.

Advantages:

- This technique addresses the issue of trajectories having different sizes (number of points in a trajectory).
- Another advantage of ERP is that it is a metric, so it satisfies the triangular inequality. This property enables triangular inequality pruning and the use of well-known spatial data indexes like R-trees and k-d trees [Bentley75].

Disadvantages:

- One of the disadvantages of the ERP distance is that it is still sensitive to measurement noise in the trajectories. This is because the ERP distance formula explicitly computes the Euclidean distance between points, as opposed to the EDR distance, that increments the distance between trajectories if the points match (are within a region).
- Another disadvantage is that it is a serial algorithm that requires significant modifications in order to take advantage of parallel architectures like GPUs.

1.2.5 Edit Distance With Projection (EDwP)

The Edit Distance with Projections (EDwP) was proposed in [RDTD+15] as a new trajectory similarity measure that, unlike all of the existing top-K trajectory query

processing techniques, specifically addresses the issue of inconsistent sampling rates. This technique can be considered as a generalization of the Edit Distance on Real Strings because it is based on the same idea of finding the least-cost set of editions that transform one trajectory into the other. The difference between EDwP and EDR is the nature of the allowed editions. The allowed editions in EDwP are: replacements and insertions. A replacement operation behaves like a matching operation, but instead of counting the cost of this operation as "1" (as EDR would do), this matching operation computes the area between the corresponding segments. An insertion, on the other hand, corresponds to projecting a point in one trajectory into the other, so in a sense this insertion would correspond to dynamically interpolating points in the trajectories. Therefore, thanks to this insertion operation, EDwP is able to address the issue of inter-trajectory sampling variations, and thanks to the insertion operation EDwP is able to address the issue of intra-trajectory sampling variations.

EDwP, just like with many other trajectory similarity measures based on editions, can be computed using dynamic programming. Moreover, the worst-case time complexity of computing the EDwP between two trajectories is $O(|p||q|)$, and the worst-case space complexity is $O(|p||q|)$.

To process top-K trajectory similarity queries, the work [RDTD+15] proposes an ad hoc indexing structure called the TrajTree, that exploits the idea of Lipschitz embedding [Bourgain85] and bounding boxes [Gutt84] to reduce the search space.

Advantages:

- The first advantage of EDwP is that it can deal with trajectories of different sizes.
- Another advantage of the EDwP technique is that it addresses the issues of local time shifting. It also addresses the issues of inter-trajectory sampling variation, intra-trajectory sampling variation and sampling phase variations.
- A final advantage of this technique is that it does not require any parameters. Hence, to implement it there is no need to perform a search in a large parameter space for an optimum parameter value.

Disadvantages:

- One disadvantage of the EDwP top-K trajectory similarity query processing technique is that it is not a distance measure, because it does not satisfy the triangular inequality. Therefore, it needs ad hoc pruning techniques and ad hoc indexing data structures to avoid implementing top-K similarity searches with an exhaustive search.
- The EDwP does not address the issue of measurement uncertainty.
- Another disadvantage of EDwP is that it does not address the issue of model uncertainty.
- This is a serial technique, so it needs substantial modifications to fully take advantage of parallel architectures like GPUs.

1.2.6 MA

Sankararaman et al. introduced in [SAMP+13] the first top-K trajectory similarity query processing algorithm, called the MA algorithm, that addresses the issue of sampling phase variations. Given two trajectories p and q, the MA algorithm finds the similarity between those two trajectories by finding a maximum score monotone assignment (also called an asymmetric monotone matching) between p and q. An *assignment between two trajectories p and q* is a pair of functions $\alpha: p \longrightarrow q \cup \{\perp\}$ and $\beta: q \longrightarrow p \cup \{\perp\}$. A point p_i (q_j) in p (q) is said to be a gap point if $\alpha(p_i) = \perp$ ($\beta(q_j) = \perp$). A gap is a maximal sequence of gap points. A *monotone assignment* is a pair of functions α and β such that if $\alpha(p_i) = q_j$, then for all $i' > i$ it is the case that $\alpha(p_{i'}) = q_{j'}$ with $j' > j$. The function β satisfies the same property.

Sankararaman et al. proposed the following function to compute the similarity score between two trajectories p and q, where $\Delta > 0$, $\lambda > 0, \theta < 0$ are chosen parameters, and $\Gamma(\alpha, \beta)$ is the set of gaps between p and q given α, β.

$$\sigma(p, q; \alpha, \beta) = \sum_{p_i \in p, \alpha(p_i) \neq \perp} \frac{1}{\lambda + \|p_i - \alpha(p_i)\|^2} + \sum_{q_j \in q, \beta(q_j) \neq \perp} \frac{1}{\lambda + \|q_j - \beta(q_j)\|^2} + \sum_{g \in \Gamma(\alpha, \beta)} (\theta + \Delta|g|)$$

The intuition behind this formula is that points p_i in p that are close to their assignments $\alpha(p_i)$ contribute very much to the score. It is a similar case with the points q_j in q that are close to their assignments $\beta(q_j)$. Additionally, the greater the number of gaps, the larger the similarity score.

The MA algorithm is similar to both DTW and Sequence Alignment [DEKM98]. The differences are the following: 1) the assignment is not necessarily symmetrical, i.e. in the associated directed matching graph, if p_i is matched to q_j, then not necessarily is q_j matched to p_i.

The MA algorithm can be computed in a way similar to the sequence alignment [DEKM98] using a dynamic programming algorithm in *O(mn)* worst-case time complexity, and in *O(mn)* worst-case space complexity. Unlike the sequence alignment algorithm, Sankararaman et al.'s algorithm uses an asymmetric matching, which forces the latter algorithm to use auxiliary recursive functions.

The authors in [SAMP+13] do not present a new pruning strategy to reduce the amount of work necessary to process top-K trajectory similarity queries. Therefore, the MA algorithm, to process top-K trajectory similarity queries, receives as inputs a query trajectory *p,* a trajectory database *db* and a non-negative integer *K*, and proceeds with an exhaustive search employing the MA-similarity measure.

Advantages:

- The MA algorithm can distinguish outliers from true trajectory deviations in virtue of its use of the gap model.

- Because the MA algorithm allows several points in one trajectory to match to a single point in the other, the MA algorithm can handle the issue of different inter-trajectory sampling rates.

Disadvantages:

- Because DTW tries to match all points in both trajectories, then when one of those trajectories exhibits strong deviations with respect to the other, the results cease being meaningful [DEKM98].
- So far, there are no pruning techniques to avoid an exhaustive search when processing top-K trajectory similarity queries, so certainly the MA-algorithm, at this time, cannot scale for Big Trajectory Data.

1.3 *Probability-based techniques*

Probability-based techniques are top-K trajectory similarity measures based on probability concepts to measure the similarity between trajectories. These techniques usually address the issues of trajectory uncertainty when processing this type of query.

1.3.1 KSQ

Ma and Lu proposed KSQ [MLSC13], which is a technique to process top-K trajectory similarity queries on uncertain trajectories. Their technique introduces a new similarity measure that specifically addresses measurement uncertainty.

To use this uncertain trajectory similarity measure it is assumed that trajectories are represented in terms of the following data model. It is assumed that each trajectory *q* in the database *db* has at each time instant *t* when the trajectory's movement was sampled an associated probability mass function (or probability density function). Then, in the time instants between consecutive sampling instants, this data model assumes a linear interpolation model. This function which dictates the probability that trajectory *q* at time *t* is located at any point *p* in the native space. Using this data model, it is then possible to define the instant p-distance of a point, the instant p-distance of a trajectory, and the interval p-distance of a trajectory.

The instant p-distance of a point is defined as follows. Given a trajectory database *db* and a query trajectory *q*, the instant p-distance of a point *x=q[t]* (the position of *q* at time *t*), denoted by $D_p(q, x, y)$ is defined as

$$D_p(q, x, t) = \sum_{y \in db} P(Q, x \succ y, t)$$

where $P(Q, x \succ y, t)$ is defined as

$$P(q, x \succ y, t) = \int \theta(Q, x, y[i]) \cdot y.pdf(y[i])\, di$$

and

$$\theta(q, x, y) = \begin{cases} 1, if\ d(x, q) > d(y, q) \\ 0, \qquad\qquad \text{otherwise} \end{cases}$$

The intuition behind this definition is that the instant p-distance from *x* to *q[t]* (*q* at time *t*) is the summation of the probabilities that all other trajectories in the database have of being closer to *q* than *x* at time *t*. Therefore, the instant p-distance of a point is always less than or equal to $|db|$.

With the definition of instant p-distance of a point, then the instant p-distance of a trajectory can be defined. Given a trajectory database *db* and a query trajectory q, the instant p-distance of a trajectory *x* in *db* at time *t*, denoted as $D_p(q, x, t)$, is defined as

$$D_p(q, x, t) = \int D_p(q, x[i], t) \cdot x.pdf(x[i])\, di$$

The intuition behind this definition is that the instant p-distance of trajectory *x* to *q[t]* is the expected value (or average) of the point p-distances from x_i to *q[t]*, where x_i are all those points that *x* might occupy (with non-zero probability) at time *t*. Just as in the case with the instant p-distance of a point, the instant p-distance of a trajectory is always less than or equal to $|db|$.

Finally, [MLSC13] proposed the interval p-distance of a trajectory. This dissimilarity measure is defined as follows. Given a query trajectory *q*, the *interval p-distance of a trajectory* *x*, denoted as $D(Q, X)_{t_e}^{t_f}$, where $[t_e, t_f]$ is *q*'s lifespan is defined as

$$D_p(q, x)_{t_e}^{t_f} = \frac{1}{t_f - t_e} \int_{t_e}^{t_f} D_p(q, x, t)\, dt$$

The intuition behind this definition is that the interval p-distance is the average of the instant p-distances from *x* to *q*. As in the other cases, it is always less than or equal to $|db|$. Ma and Lu [MLSC13] have proved that one way of thinking of the interval p-distance of a trajectory is as the expected rank of the trajectory if the trajectories in the database are sorted in decreasing order of similarity from *q*, where the highest ranked trajectory is the least similar trajectory to *q*.

Ma and Lu propose in [MLSC13] a trajectory index to process top-K trajectory similarity queries. This index consists of a uniform spatial grid placed in the spatial dimension of trajectories. If a trajectory in the database intersects multiple grid cells, then that trajectory is split at the grid cell boundaries and the resulting sub-trajectories or segments are "inserted" into their corresponding grid cells. In each grid cell there is an R-tree to index (according to the temporal dimension) the segments in that grid cell.

To process a top-K trajectory similarity query, KSQ associates with each grid cell a lower bound and an upper bound to the KSQ scores of the trajectories that are wholly contained in each grid cell. KSQ uses a min-heap in which it inserts all the grid cells that are close to the query trajectory q, and the key for this min-heap is the lower bound to the KSQ scores of trajectories contained within that cell. Once KSQ has inserted these cells in the min-heap, it enters a while loop in which it takes the top cell at the min-heap. If this cell has a lower bound that is greater than the K-th smallest upper bound of the KSQ scores seen so far, then it knows that all the trajectories that intersect that cell are pruned. This is because, by definition, we have seen K trajectories that have better scores than any of the trajectories intersecting the cell in question. Otherwise, KSQ updates the lower and upper bounds of the KSQ scores of the segments contained in that cell. If the trajectories associated with those segments have a lower bound that is greater than the K-th smallest upper bound of the KSQ scores seen so far, then that trajectory can be pruned. When the while loop ends, KSQ obtains a set of candidate trajectories, which contains the true result set. Then, KSQ must compute the exact KSQ

scores for all the candidate trajectories, then sort them in increasing order of KSQ scores, and then return the set of trajectories with the smallest KSQ scores.

Advantages:

- By considering a probability mass function at each sample point of each trajectory, and by incorporating that information into the trajectory similarity measure, the KSQ top-K trajectory similarity query processing technique addresses the issue of measurement uncertainty.

Disadvantages:

- The KSQ top-K trajectory similarity query processing technique does not address the issue of model uncertainty. This is because the trajectories are still sampled at discrete time intervals, and thus still relies on the interpolation model of trajectories.
- This technique is a serial technique, so it requires a significant research effort to ensure that it works efficiently on parallel architectures like GPUs.

1.4 *Feature Comparison of Top-K Trajectory Similarity Query Processing Techniques*

We have presented in Section 1 a discussion of state-of-the-art techniques for top-K trajectory similarity query processing techniques. None of those techniques addresses the all the issues identified in Section 3. In particular, none of those techniques provides

support for different trajectory sizes, measurement uncertainty, model uncertainty, triangular inequality, and support for Big Trajectory Data. To fill this gap, we introduce a novel system to process top-K trajectory similarity queries in parallel on Big Data using GPUs that is capable of handling both certain and uncertain trajectory data. Our system addresses the issue of support of different trajectory sizes due to its similarity measure, the Hausdorff distance, which can compute the similarity between two trajectories of different sizes. Our proposed system addresses the issues of measurement and model uncertainty thanks to its construction of trajectory data models that are then used to estimate the true path of the trajectories. This system also addresses the issue of support for parallel processing because it is designed to run on GPUs and multicore CPUs by addressing the GPU issues discussed in Section 2.4.4.

Our proposed system consists of four novel algorithms: TKSimGPU to process top-K trajectory similarity queries; Top-KaBT to reduce the size of the candidate set generated by top-K trajectory similarity query algorithms; TrajEstU to estimate the true trajectory when data uncertainty exists; and TraclusGPU to perform local trajectory clustering to aid in the preprocessing stage of TrajEstU. TKSimGPU works by iteratively processing near-join similarity queries, while Top-KaBT calculates the lower and upper bounds of the Hausdorff distance between candidate pairs, and then uses these bounds to remove spurious candidates. Top-KaBT exploits GPUs to improve TKSimGPU by ensuring load balancing across the threads, ensuring memory coalescing, and using special pruning techniques that reduce the size of the candidate set. TrajEstU splits the lifetime of an object's trajectory into time intervals where the object's acceleration is nearly

constant. Then TrajEstU uses the local trajectory clusters to obtain the movement patterns that are prevalent in the areas where trajectories have low-sampling rates, and uses linear regression to fit a constant acceleration model to the observed positions of the moving object. Finally, TraclusGPU helps TrajEstU scalably find those local trajectory clusters that are used in the construction of trajectory models.

Table 2 presents a feature comparison of the top-K trajectory similarity query processing algorithms reviewed in Section 1. If a cell contains the word "Yes," that indicates that the technique referred to in that row addresses the issue listed at the top of that column. On the other hand, if a cell contains the word "No," that indicates that the corresponding issue is not addressed by that technique.

	Local time shifts	Different trajectory Sizes	Sampling rate variations			Dimension of the space of parameters	Uncertainty		Parallel processing
			Inter-trajectory	Intra-trajectory	Phase		Measurement uncertainty	Model uncertainty	
Euclidean [FRM94]	No	No	No	No	No	No	No	No	No
Hausdorff [NJS11]	No	Yes	Yes	No	No	Yes	No	No	No
DTW [BK94]	Yes	Yes	No	No	No	Yes	No	No	No
LCSS [VKG02]	Yes	Yes	No	No	No	No	No	No	No
ERP [CN04]	Yes	Yes	No	No	No	No	No	No	No
EDR [COO05]	Yes	Yes	No	No	No	No	No	No	No
DISSIM [FGT07]	Yes	Yes	No	No	No	Yes	No	No	No
wDF [DTS08]	Yes	Yes	No	No	No	No	No	No	No
MA [SAMP+13]	No	Yes	No	No	Yes	No	No	No	No
KSQ [MLSC13]	No	No	No	No	No	Yes	Yes	No	No
EDwP [RDTD+15]	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
TKSimGPU +Top-KaBT	No	Yes	Yes	No	No	No	No	No	Yes
TKSimGPU +Top-KaBT + TrajEstU	No	Yes	Yes	No	No	No	Yes	Yes	Yes

Table 2. Feature comparison of top-K trajectory similarity query processing techniques

2 Literature Review of Techniques for Estimating Uncertain Trajectories

As we have seen in Chapter I, when discussing the issues that need to be addressed by top-K trajectory similarity query processing techniques, if a moving object is equipped with a location sensor, such as a GPS, the it is possible to periodically sample the movement of the object and store the resulting sequence of locations occupied by the object as a trajectory. However, all sensors, including GPS [GPS17], have an inherent measurement or observation error. In addition to this measurement error, there is the fact that if we model the movement of an object as a trajectory, then the model itself would likely incur a model error because the trajectory could need to be sampled at an infinite rate to perfectly describe the movement of an object. As has been discussed in Chapter I, a new scalable and accurate top-K trajectory similarity query processing technique should address these two issues (measurement and model uncertainty).

Since our proposed techniques involve reducing the uncertainty of trajectories, this chapter presents a survey of techniques designed to address the problem of estimating the true path of the moving object (which is a dynamical system) at every time instant using a sequence of uncertain measurements (the points making up the trajectory). The existing techniques are then classified according to whether they make use of an underlying trajectory database or not.

2.1 *Techniques that do not exploit a database of trajectories*

The following techniques do not make use of a database of trajectories in order to improve the quality of their estimates.

2.1.1 Mean and Median Filter

The mean and the median filters are techniques that use measurements z_i about a system to estimate its true internal state x_k at time k, when the measurements are noisy. The idea behind the mean filter is to store the previous measurements in a window of size *n* and, even if the measurements of the system are noisy, we can average all these measurements in this window to produce a less noisy estimation of the internal state. So, the mean filter works by computing $\widehat{x_k} = 1/n \sum_{j=k-n+1}^{k} z_j$ as an estimator for x_k. The disadvantage of the mean filter is that it is very sensitive to outliers. To address this problem, the median filter works similarly as the mean filter in keeping a window of size *n* to store the previous measurements of the system, but instead of computing the average of the window measurements it computes its median to produce an estimator $\widehat{x_k} = median(z_{k-n+1}, \dots, z_k)$.

Advantages:

- One of the biggest advantages of both the mean and the median filter are their simplicity.

Disadvantages:

- The mean filter is very sensitive to outliers because even a single outlier can perturb the true path estimate for a trajectory.
- This technique does not produce an estimate of the uncertainty of its predictions.

2.1.2 Kalman Filter

Kalman proposed in [K60] a technique, that today is known as the Kalman Filter, that can estimate the true internal state x_k at time k of a linear dynamical system using a sequence of uncertain observations $\{z_i | i \geq 0\}$. Kalman Filters can be applied to the estimation of the true path of a moving object (moving in unconstrained space) if we consider the system to be the moving object itself, and if we consider the sequence of points of its associated trajectory as the set of (uncertain) observations of the state of the system.

The classical Kalman Filter requires that we have two models: a linear model describing how the system moves from one state to another in time (how the object moves as a function of time), called the *state transition model*, and a model describing how the system observations (which in our trajectory application correspond to the points of a trajectory) relate to the internal state of the system (that is how the GPS model reports the positions given the true state of the moving object). The Kalman Filter's state transition model is of the form: $x_k = F_k x_{k-1} + w_k$, where $x_k \in \mathbb{R}^n$ is the state of the system at time *k* (i.e. the true position of the moving object at time *k*), $F_k \in \mathbb{R}^{n \times n}$ is the state transition matrix at time *k* indicating how the object moves from state x_{k-1} to state x_k, $w_k \in \mathbb{R}^n$ is the process noise; this noise arises because the state transition matrix may not be able to accurately capture the exact nature of the behavior of the system, so that noise accounts for this uncertainty. Besides the state transition model, there is an observation model that relates the internal state of the system $x_k \in \mathbb{R}^n$ at time *k* with the actual observations made. This observation model is of the form: $z_k = H_k x_k + v_k$,

where $z_k \in \mathbb{R}^m$ is the observation at time k (i.e., the GPS measurement of the position of the object at that time), $H_k \in \mathbb{R}^{m \times n}$ is the observation matrix, and $v_k \in \mathbb{R}^m$ is the observation noise; this observation noise is introduced because the observation matrix may not be able to accurately capture the exact nature of the relationship between of the observations and the true state of the system, so that this noise accounts for the uncertainty.

The algorithm for the Kalman Filter [LLD06] receives as inputs $E(x_0) \in \mathbb{R}^n$ (the expected value of the internal state of the system), $Q \in \mathbb{R}^{m \times m}$ is the covariance matrix of the model error, $P_0 \in \mathbb{R}^{n \times n}$ is the covariance matrix of initial state, and the output is an estimator $\widehat{x_k} \in \mathbb{R}^n$ of the state of the system at time *k.* The Kalman filter then proceeds as follows:

1. Give an estimation $\widehat{x_0}$ for the initial state of the system at time 0: $\widehat{x_0} = E(x_0)$, and an estimation for the covariance of the estimator at time 0: $\widehat{P_0} = P_0$.
2. Model Forecast step: Use the estimation of the system state at time *k-1:* $\widehat{x_{k-1}}$ and the state transition model to obtain a model forecast $x_k^f = F_k \, \widehat{x_{k-1}}$, and compute its associated forecast covariance P_k^f.
3. Data assimilation step: Use the observation z_k of the system at time k to improve upon the model forecast x_k^f obtained in the previous step, by obtaining the estimation $\widehat{x_k}$ and its associated covariance matrix $\widehat{P_k}$. Then go back to step 2 to proceed to obtain estimates for future timestamps.

Advantages:

- The Kalman Filter is a linear minimum variance estimator [LLD06]. This means that the Kalman Filter is optimum in the sense that among all linear estimators for the true path of the query object, no other estimator has smaller variance.
- The Kalman Filter provides an estimation of the uncertainty associated with the prediction of the true path of the object. This advantage makes Kalman filters particularly special because they do not only estimate the true path of the object, but they also inform about the uncertainty of such estimate.

Disadvantages:

- One of the disadvantages of the Kalman Filter is that since each moving object is different, then so are the associated dynamical models. This means that to estimate the true path of a very large number of objects, then that would require fitting a very large number of models (at least one per object).

2.1.3 Particle Filter

The particle filter [Gor94] is another technique used to estimate the internal state of a system based on observations of such system. The difference between the classical Kalman filter and the particle filter is that the latter relaxes the condition that the dynamics of the moving object has to linear (just as there are non-linear Kalman filters, which are extensions of the classical Kalman filter); thus, the particle filter can accommodate more general movement dynamics, at the expense of a higher time complexity algorithm.

The algorithm for the particle filter receives as inputs a probability distribution $p(z_i|x_i)$, which indicates the probability of observing z_i if the true state of the system is x_i for every possible observation value and every possible state, then, the Particle filter proceeds as follows:

1. Generate a random set of particles x_k.
2. (Model forecast) Use the dynamic model $x_i^f = p(x_i|x_{i-1})$ to simulate how the particles move. This is the analog of the model forecast step in the Kalman filter.
3. (Importance weights computation) Compute the importance weight of each particle $x_i^f = p(z_i|x_i^f)$, and normalize these weights to 1. The weight of a particle says how likely it is to observe what has been observed, given that the particle is the true state of the system. Therefore, particles with larger weights are more consistent with the observations.
4. Select a new set of particles at random from the set of particles already created. The probability of selecting a given particle is proportional to its weight.
5. Go back to step 2 and repeat until producing the estimations for the desired time.

Advantage:

- The particle filter can be made to work on network roads and paths (constrained spaces). This is an advantage of particular significance because some techniques work only on network roads (e.g., HRIS [ZZXZ12]).

Disadvantages:

- The particle filter in general takes a significantly longer computation time than the classical Kalman filter [ZZ11] in order to produce its estimates.

- To estimate the true paths of each trajectory in a large database, the particle filter requires constructing a different model for each object because objects may very likely follow different dynamics.

2.2 *Techniques that exploit a database of trajectories*

The techniques that we now describe have all in common that they take advantage of a database of trajectories in order to improve the accuracy of their estimates. However, these techniques make the tacit assumption that the input trajectory follows the same dynamical model of most of the trajectories in the database, which may not be the case.

2.2.1 HRIS

Zheng et al. [ZZXZ12] address the problem of reducing the uncertainty of low-sampling rate trajectories, i.e., trajectories such that the average interval between consecutive points is over 2 mins, in road networks (constrained space). To accomplish this, they propose an algorithm, called HRIS, that fills the low-sampling rate sections of a trajectory by searching for other nearby trajectories (called *reference trajectories*) in the database that satisfy certain network constraints, like velocity constraints. After performing this search, their algorithm finds a set of associated road network paths corresponding to those reference trajectories that maximize an objective function based on popularity and uniformity of the traffic through that road network path. This work, unlike our work, assumes that objects move in a constrained space, so it exploits the knowledge provided by a road network to reduce the uncertainty in the low-sampling rate trajectories. Therefore, it is not applicable in the scenario of objects moving in

unconstrained space (like hurricanes and animal trajectories), where objects do not move on road networks, or where there is no road network available.

The HRIS algorithm receives as inputs a low-sampling rate trajectory q whose true path across a road network is to be estimated, a road network *RN,* and a database *DB* of trajectories moving. The output is a path on the road network that accurately describes the true path of the object. The HRIS algorithm works as follows:

1. For every consecutive pair of points q_i and q_{i+1} in q, it searches for a set of *reference trajectories RT(i,q)*. Reference trajectories are trajectories in *DB* that are close to both points, q_i and q_{i+1}, that do not exceed the maximum speed allowed in the road network.
2. (Local Route Inference) In this step, HRIS seeks to infer what are the possible routes in the road network *RN* that the moving object associated with q could have taken when moving from point q_i and q_{i+1}. Therefore, this step is run for every pair of consecutive points of q. For this pair of points q_i and q_{i+1}, HRIS builds a directed graph called the *traverse graph*, whose nodes are the edges of the road network *RN* such that there exists at least one trajectory in *RT(i,q)* that travels close enough (HRIS requires a tolerance parameter to determine this) to such edge in the road network. These edges in the road network that are close to trajectories in *RT(i,q)* are called *traverse edges*. Now, there is an edge from node n_1 to node n_2 in the traverse graph if the road network edge n_2 is within λ hops from the road network edge n_1 ($\lambda > 0$ is another parameter). Once the traverse graph for q_i and q_{i+1} is built, HRIS runs a top-K shortest paths algorithm

[Yen71] (K is an input parameter to HRIS) on the traverse graph starting from the node on the traverse graph that corresponds to the closest road network edge to q_i to the node on the traverse graph that corresponds to the closest road network edge to q_{i+1}. The resulting set of top-K shortest paths is the set of possible routes that q could have taken from q_i to q_{i+1}.

3. (Global Route Inference). In this step, HRIS seeks to infer the most likely routes that q could have taken at all points. To do this, HRIS has already for every pair of consecutive points q_i and q_{i+1} an associated set of K possible road network paths, and needs to connect these paths together to obtain a global possible road network path that q could have likely taken. HRIS assigns to every possible road network path of every pair of q a weight proportional to the number of reference trajectories that traverse that path. The higher the weight, the more popular the route; hence, the more likely, according to HRIS, that route is.

   If R_a is a local route obtained from Step 2, that q could likely have taken when moving from point q_i to q_{i+1}, and R_b is also a local route but q could have likely taken it when moving from point q_{i+1} to q_{i+2}, then the strength of the connection between R_a and R_b is proportional to the number of reference trajectories that travel both on R_a and R_b. To perform global route inference, HRIS solves an optimization problem wherein it searches for the road network path with the highest score, and the score of a global route $R=(R_1,R_2,\ldots,R_n)$, with the R_i being local routes, is computed as $score(R) = f(R_1) \cdot g(R_1, R_2) \cdot f(R_2) \cdot \cdots f(R_m)$, where $f(R_i)$ is the score of the local route R_i and $g(R_i, R_{i+1})$ is the score of the connection between R_i and R_{i+1}. This optimization problem is solved through

dynamic programming.

Advantages:

- One of the advantages of HRIS is that it exploits the knowledge contained in a database of trajectories in order to estimate the true path of an input trajectory. This is particularly advantageous when the input trajectory obeys the same dynamical model followed by many of the trajectories in the database.

Disadvantages:

- A disadvantage of this technique is that it can only be applied for objects moving in constrained networks. Therefore, for trajectory applications like finding birds with similar migration patterns to a given bird species, or finding similar hurricane trajectories this technique is not applicable because neither birds nor hurricanes have constrained movements.
- Another disadvantage of HRIS is that when performing local route inference, it needs to run the top-K shortest paths algorithm on the traverse graph. However, the complexity of this algorithm is $O\left(K \cdot |V| \cdot (|E| + |V| \cdot \log(|V|))\right)$, which does not scale well for large graphs.

2.2.2 Chazal et al.'s Algorithm

Chazal et al. present in [CCGJ+11] a trajectory smoothing technique to reduce the noise in a trajectory and this help estimate the true path of the object. This technique receives as input a database D of trajectories, an uncertain trajectory q whose true path we want

to estimate, and a positive integer $K > 0$. The technique then proceeds to use the knowledge in the database to help estimate the true path of the object. To do so, the technique proceeds as follows:

1. Embed the input trajectory q and each trajectory in the database D into a $2\times(2n+1)$-dimensional space (where n is the dimension of the space where each point of each trajectory lies. Usually $n=2$) by assigning trajectory $p= (h_a, h_{a+1,\dots,} h_{a+m})$ (such that each of its points lies in n-dimensional space) to the trajectory $p' = (h_{i-n}^{(x)}, h_{i-n}^{(y)}, h_{i-n+1}^{(x)}, h_{i-n+1}^{(y)}, h_{i+n}^{(x)}, h_{i+n}^{(y)})$ for i in $\{a,a+1,\dots,a+m\}$ (such that each of its points lies in $2\times(2n+1)$-dimensional space). If we choose $n=1$, then the sequence $s = ((1,1), (2,2), \dots (k,k))$ is mapped to $((1,1,2,2,3,3), (2,2,3,3,4,4),\dots, (k\text{-}2, k\text{-}2, k\text{-}1,k\text{-}1,k,k))$. Let us call the set of all points in $2\times(2n+1)$-dimensional space resulting from embedding the trajectories in D as the set of high-dimensional points.
2. For every point q_i' in the embedded trajectory q', find its k nearest neighbor points belonging to the set of high-dimensional points, and move q_i' to the average of its k nearest neighbor points.
3. For every trajectory p' that resulted after steps 1 (embedding) and 2 (moving each point toward the average of its nearest neighbors), we recover a trajectory in the original n-dimensional space by taking the middle n-coordinates of each of its points. For example, if p' has points ($(1,1,2,2,3,3), (2,2,3,3,4,4)$) living in the embedded high-dimensional space, we can recover from this trajectory a corresponding trajectory, called *estimated trajectory*, in the original n-

dimensional space by taking the middle n-coordinates: *((1,1,**2,2**,3,3), (2,2,**3,3**,4,4))* → *((2,2),(3,3))*.

4. The final true path estimation for q is the estimated trajectory obtained in step 3.

Advantages:

- One of the main advantages of the Chazal et al.'s algorithm is its simplicity.

Disadvantages:

- One disadvantage of this technique is that computing the k nearest neighbors of every point of the query trajectory in a space with $2\times(2n+1)$ dimensions has worst-case time complexity $O(|inputTrajectory| \times |DB|)$.
- Another disadvantage is that if we store in memory the higher-dimensional embedding of the database, then the worst-case space complexity increases by a factor of $2n+1$. However, if the trajectories in the DB are stored un-embedded, then that increases the time complexity of the algorithm.

2.3 *Feature Comparison of Techniques for Estimating Uncertain Trajectories*

We have presented in Section 2 a discussion of state-of-the-art techniques for estimating uncertain trajectories. In the previous discussion we presented a group of techniques, consisting of the mean/median filter, Kalman filter, and particle filter, that does not make use of a database of trajectories to produce estimates for an input trajectory with uncertainty. This is a disadvantage of these techniques because it is often the case that an input trajectory has similar dynamics and behavior to that of other trajectories in a

database. For example, if the database contains bird migration trajectories, and the input trajectory corresponds to another bird of the same species, then it is very likely that the input trajectory will behave similarly to the trajectories in the database. Therefore, it is possible to exploit this knowledge to improve the accuracy of the true path estimation of the input trajectory. In this previous discussion we also presented another group of techniques, consisting of HRIS and Chazal et al.'s algorithm, that do make use of a database of trajectories to improve the quality of their estimates. However, HRIS works only in constrained spaces, so it is not applicable for unconstrained spaces. Chazal et al.'s algorithm works for unconstrained spaces (therefore, it also works in constrained spaces), but it has the disadvantage that when estimating the true path of an input trajectory, it requires expensive k-nearest neighbor searches in a high-dimensional space for every single point of the input trajectory. In addition to this, Chazal et al.'s algorithm is by design a serial technique, so it requires substantial modifications in order to run on parallel architectures like GPUs.

As a conclusion of our above discussion, we see that none of the presented techniques satisfies all of the following desirable properties: support for both constrained and unconstrained spaces, exploitation of a database of trajectories, and support for parallel processing. To address this gap, we proposed an innovative algorithm called TrajEstU, (one of the algorithms that make up our proposed system), which does satisfy all these desirable properties. TrajEstU works by locally clustering the trajectories in a database to obtain the local behavior patterns around the input trajectory. This clustering phase is performed off-line and only once per database. After this is done, TrajEstU splits the

input trajectory into near-constant acceleration intervals. At each of these near-constant acceleration intervals, TrajEstU fits a linear regression models that takes into consideration the local behavior patterns found in the trajectory database. Unlike Chazal et al.'s algorithm, TrajEstU does not require expensive k-nearest neighbor searches in a high-dimensional space for every single point of the trajectory, which can lead to expensive performance penalties when dealing with Big Trajectory Data. Moreover, TrajEstU has support for parallel processing because its most computationally expensive phase (the local trajectory clustering) is designed for running on GPUs.

CHAPTER III
PROPOSED SYSTEM AND TECHNIQUES

This chapter first presents an overview of our proposed system and techniques, and then an in-depth discussion about them. Our proposed techniques are the following: TKSimGPU for processing top-K trajectory similarity queries on GPUs; Top-KaBT for pruning the candidate of top-K trajectory similarity queries on GPUs; TrajEstU for estimating the true path of uncertain trajectories; and TraclusGPU for local trajectory clustering on GPUs.

1 Overview of the proposed system and techniques

In this dissertation we propose a novel system to process top-K trajectory similarity queries in parallel on Big Data using GPUs. The system is capable of handling both certain and uncertain trajectory data. The system consists of four novel techniques. The first one, TKSimGPU, is a top-K trajectory similarity query processing algorithm for GPUs. TKSimGPU is a trajectory query processing algorithm based on the filter-and-refine approach, which consists of an initial filter stage (which is cheap in terms of execution time) in which a candidate set of trajectory pairs is generated, and a later refine stage (a more expensive stage), in which this candidate set is examined more thoroughly in order to find the true query result set.

The second proposed technique Top-KaBT, is a parallel GPU pruning technique to reduce the number of spurious candidate pairs *(p,q)* generated by top-K trajectory similarity query techniques using similarity measures that satisfy the triangular inequality. Top-KaBT was proposed because even though TKSimGPU represented an

efficient and scalable algorithm, it still could potentially generate a large number of spurious candidate pairs in its filter stage, which led to a large amount of unnecessary computations in its refine stage. The purpose of Top-KaBT is then to remove these spurious candidate pairs, thereby reducing the associated performance penalty. To accomplish this, Top-KaBT is run in between TKSimGPU's filter and refine stages. We call this resulting algorithm TKSimGPU + Top-KaBT.

So far we have mentioned that the TKSimGPU + Top-KaBT can deal with top-K trajectory similarity queries. However, trajectories can be uncertain, and this can have negative impacts on the accuracy of the queries. For this reason, we proposed a third technique, called TrajEstU, to reduce the negative impacts of uncertainty on the accuracy of trajectory similarity queries. TrajEstU has two phases: a pre-processing stage (more expensive in terms of execution time), and an online stage (very cheap in terms of execution time). The idea is that TrajEstU's pre-processing stage is run on the trajectory database (Q) before any query processing takes place. Then, when the top-K trajectory similarity queries arrive, TrajEstU's online stage is run on each query before passing the resulting trajectory

In our experimental evaluation we noticed that TrajEstU's online stage had a negligible execution time (even on serial processors); however, its pre-processing stage was too expensive in terms of computational time, which could affect TrajEstU's practicality when dealing with Big Trajectory Data. For this reason, we proposed a fourth technique, called TraclusGPU, which performs TrajEstU's pre-processing stage

(consisting of local trajectory clustering) on a GPU. In Figure 17 we present a diagram representing the relationships between our proposed techniques. Figure 18 contains the pseudocode of the overall system.

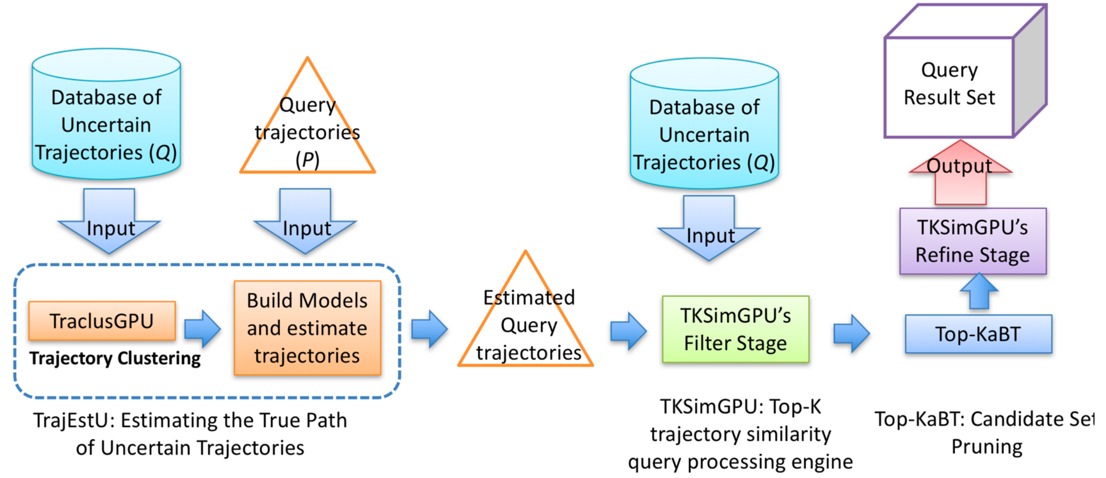

Figure 17. Workflow of our proposed system

Algorithm Top-K Trajectory Similarity Queries

Input: Two sets P and Q of trajectories. Integers $K > 0$ and $Q_sample_size \leq |Q|$. A real number $\xi > 1$.

Output: A set $Result$ containing the result of a top-K trajectory similarity query.

// Produce better estimates for the trajectories

1: **if** Trajectories are uncertain **then**

2:     **for** $query \in P$ **do**

3:         $query \leftarrow TrajEstU(query, Q)$

4:     **end for**

5: **end if**

// Call the filtering stage of TKSimGPU

6: $candidates \leftarrow Filter_TKSimGPU(K, P, Q, Q_sample_size, \xi)$

// Call Top-KaBT to prune the candidate set

7: $candidates \leftarrow Top - KaBT(K, candidates)$

// Call the filtering stage of TKSimGPU

8: $result \leftarrow refine_TKSimGPU(K, candidates)$

9: **return** $result$

Figure 18. Overall algorithm

2 TKSimGPU: A GPU technique for Top-K Trajectory Similarity Query Processing

In this section we present a parallel algorithm, called *TKSimGPU*, for top-K trajectory similarity queries, and discuss how to implement it on a GPU.

2.1 *Motivation of TKSimGPU*

As we have mentioned in Section I.2.3, top-K trajectory similarity query processing techniques have a wide range of applications stemming from biology, bioimaging, to social networks, etc. However, processing this type of queries poses significant computational challenges stemming from the sizes of the datasets, the sizes of the trajectories, and the computational complexity of the trajectory similarity measure itself. One strategy that can be used to tackle these challenges is the use of parallel computer architectures such as GPUs.

GPUs are co-processors installed on most computers (mobile devices, desktops, workstations, supercomputers, etc.) to render graphics, but that can also be used for general purpose parallel programming. Besides being widely available, GPUs are very energy efficient, and on certain kinds of algorithms they can perform up to an order of magnitude of higher single-precision floating point instruction throughput than the best multicore chips available. All these reasons make GPUs an ideal architecture with which to tackle the computational issues of top-K trajectory similarity query processing algorithms.

For these reasons, we have proposed a parallel GPU algorithm to scalably process top-K trajectory similarity queries that addresses all the issues of GPUs, low global memory bandwidth relative to the number of threads (global memory coalescing), low PCIe bandwidth, efficient use of shared memory banks, thread divergence, and load balancing, which were discussed in Chapter I.

2.2 *Overview of TKSimGPU*

As described in Section I.2.3, the top-K trajectory similarity query takes a positive integer K and two sets P and Q of trajectories, and finds for every trajectory $p \in P$ the set of K Q-trajectories most similar to p. Our proposed parallel GPU algorithm for top-K trajectory similarity queries uses the filter-and-refine processing scheme [JS07], which consists of two steps: the filter and the refine steps. The filter step selects for every $p \in P$ a candidate set $C_p \subseteq Q$ with $|C_p| \geq K$, such that the K Q-trajectories most similar to p belong to C_p. The refine step takes C_p and then computes the actual similarities between p and every candidate in C_p and returns the K most similar trajectories. The idea behind this scheme is to avoid exhaustively finding the similarities for every pair $(p, q) \in P \times Q$. This is accomplished by having the filter step cheaply prune away many trajectories that surely will not form part of the result set, and then having the refine step actually compute the exact similarity measures between p and every element in C_p.

2.3 *The TKSimGPU Algorithm*

The idea behind TKSimGPU is that a top-K trajectory similarity query can be answered by performing successive filtering steps of the near-join trajectory similarity query with decreasing ε-range values (the near-join trajectory similarity query finds all those trajectories with a similarity at least ε) until every trajectory $p \in P$ has at least *K* most similar trajectories in *Q*. An example of this is the following. If we want to find the *K Q*-trajectories most similar to $p \in P$ we first perform a filter step of the near-join similarity query with an initial range value of ε > 0. This step will return a subset $C_p \subseteq Q$ such that the Hausdorff distance from p to every $q \in C_p$ is not greater than ε. If $|C_p| < K$, we need to repeat, or restart, the filter step with a larger ε, and we proceed in this manner until $|C_p| \geq K$. Once we are certain that every $p \in P$ has at least *K* most similar trajectories in *Q*, we can select the *K Q*-trajectories most similar to *p*. This idea is similar to one of the strategies used to answer kNN queries on point data by doing successive range queries with different radii [BCG02]. The issue when using this strategy is to try to choose a large enough ε > 0 for the near-join trajectory similarity, with the intention of reducing the total number of restarts that need to be performed, and at the same time choosing this ε small enough so as to avoid a situation where the set C_p is almost *Q*, for any $p \in P$.

Figure 19 presents a pseudo-code description of our algorithm. In Line 29 we obtain, without replacement, a random sample Q_sample of size Q_sample_size of trajectory identifiers from *Q*.

Algorithm TKSimGPU

Input: Two sets P and Q of trajectories. An integer $K > 0$. An integer $\sigma \leq |Q|$. A real number $\xi > 1$.

Output: A set $Result \subseteq P \times Q \times \mathbb{R}$ such that $Result = \cup_{p \in P} C_p$, where, $|C_p| = K$ and for every $p \in P$, and $q \in Q$ that is among the K most similar Q trajectories to p, $(p, q, hausd(p,q)) \in C_p$

1: **function** TOP-K TRAJECTORY SIMILARITY(K, P, Q, σ, ξ)
2: $PQ_candidates \leftarrow$ FILTER_TKSIMGPU(P, Q, K, σ)
3: $Result \leftarrow$ REFINE_TKSIMGPU($K, PQ_candidates$)
4: **return** $Result$
5: **end function**

6: **function** ESTIMATE_ϵ($K, P_sample, Q_sample, prev_\epsilon, \xi$)
7: $KNearest = \emptyset$
8: **for** $p \in P_sample$ **do in parallel**
9: Find the list of pairs $L_p = [(hausd(p,q), q) \mid q \in Q_sample]$
10: $S_p \leftarrow$ Sort L_p, in parallel, in increasing order of $hausd(p,q)$
11: $KNearest \leftarrow KNearest \cup \{S_p[K].fst\}$
12: **end for**
13: $\epsilon \leftarrow$ Using a parallel reduction, find the average of $KNearest$
14: **if** $\epsilon \leq prev_\epsilon$ **then return** $\xi \cdot prev_\epsilon$
15: **else return** ϵ
16: **end if**
17: **end function**

18: **function** NEAR-JOIN FILTER($P, Q, \epsilon, \mathcal{G}$)
19: $P_raster \leftarrow \{(p_t, c) \in Tracks(P) \times \mathcal{G} \mid \text{eMBR}(p_t, \epsilon) \text{ intersects } c\}$
20: $Q_raster \leftarrow \{(q_t, c) \in Tracks(Q) \times \mathcal{G} \mid \text{eMBR}(q_t, 0) \text{ intersects } c\}$
21: $TrkC \leftarrow \{(p_t, q_t) \mid \exists c : (p_t, c) \in P_raster, (q_t, c) \in Q_raster\}$
22: $TrkCandidates \leftarrow$ remove duplicates, in parallel, from $TrkC$
23: $TrajC \leftarrow$ TRACKSTOTRAJECTORIES($TrkCandidates$)
24: $TrajCandidates \leftarrow$ remove duplicates, in parallel, from $TrajC$
25: **return** TrajCandidates
26: **end function**

27: **function** FILTER_TKSIMGPU(P, Q, K, σ)
28: $P_sample \leftarrow P$
29: $Q_sample \leftarrow get_sample(Q, \sigma)$
30: $\epsilon \leftarrow 0$
31: **do**
32: $\epsilon \leftarrow estimate_\epsilon(K, P_sample, Q_sample, \epsilon, \xi)$
33: $PQ_candidates \leftarrow near_join_filter(P, Q, \epsilon)$
34: $D \leftarrow \{(p, cnt) \mid p \in P, cnt = |\{(p,q) \in PQ_candidates \mid q \in Q\}|\}$
35: $Incomplete \leftarrow \{p \mid \forall (p, m) \in D : m < K\}$
36: **if** $Incomplete \neq \emptyset$ **then**
37: $P_sample \leftarrow Incomplete$
38: $Q_sample \leftarrow get_sample(Q, \sigma)$
39: **end if**
40: **while** $Incomplete \neq \emptyset$
41: **return** $PQ_candidates$
42: **end function**

43: **function** REFINE_TKSIMGPU($K, PQ_candidates$)
44: **for** $p \in \Pi_P(PQ_candidates)$ **do in parallel**
45: $C_p \leftarrow \{(p, q, hausd(p,q)) \mid (p,q) \in PQ_candidates\}$
46: $Sorted_p \leftarrow$ sort C_p in increasing order of $hausd(p,q)$
47: $Result_p \leftarrow Sorted_p[0, \ldots, K-1]$
48: **end for**
49: Using a parallel reduction compute $Result \leftarrow \cup_{p \in P} Result_p$
50: **return** $Result$
51: **end function**
52: **return** TOP-K TRAJECTORY SIMILARITY(K, P, Q, σ, ξ)

Figure 19. TKSimGPU algorithm

For example, if $P = \{1,4\}$, $Q = \{6,7,9\}$, $K = 2$, $sample_size = 2$, then $Q_sample = \{7,9\}$ is a random trajectory identifier sample.

Lines 31 through 40 implement the filtering step mentioned before. Using the two sets of trajectory identifier samples we can find for every $p \in P_sample$ the Hausdorff distance to the K-th closest trajectory to p, and use the average, over all $p \in P_sample$, of these values as an initial estimate for ε. Continuing our example above, suppose that $hausd(1,7) = 1.2$, $hausd(1,9) = 2.1$, $hausd(4,7) = 3.1$ and $hausd(4,9) = 3.0$, then the 2nd closest Q-trajectory ($K = 2$) to 1 is 9, and the 2nd closest Q-trajectory to 4 is 7; and our initial estimate for ε will be $\varepsilon = (hausd(1,9) + hausd(4,7)) / 2$. Once we have the value of ε we perform a near-join filter in Line 33, after which we obtain a set *PQ_candidates* consisting of all those pairs of identifiers $(p, q) \in P \times Q$ indicating that *q could* form part of the K Q-trajectories most similar to p. In Line 34 we count, for every $p \in P$, the number of candidates (which are Q trajectories) found for p. For example, if after the near-join filter we obtain *PQ_candidates* = *{(1,7), (1,9), (3,7)}*, then after Line 34 we would obtain the set $D = \{(1,2), (3,1), (4,0)\}$, indicating that trajectory 1 has two pairs, trajectory 3 has 1 pair and trajectory 4 has no pair in *PQ_candidates*. Once we know how many candidates have been found for every $p \in P$, we are interested in those $p \in P$ for which we have found fewer than K candidates; these will be the elements forming part of the set *Incomplete*. In our example, *Incomplete* = {3} because 3 does not have 2 candidates. The set *Incomplete* will be used in the next iteration, in Line 32, to estimate a larger new ε.

Lines 6 through 17 contain the details of how to select ε values that will be used for the near-join similarity filters. The idea is that with P_sample and Q_sample we can compute, for every $p \in P_sample$, at what distance d_p is the *K*-th most similar *Q_sample*-trajectory of p. Later in Line 13 we average all the d_p values. The idea is that the average of the d_p should be an estimate of the average distance at which the *K*-th most similar *Q*-trajectory is from every $p \in P$. Now, since we may be taking different samples *P_sample* and *Q_sample* at every iteration of the do-while in Line 31, it could be the case that the sequence of ε values that we obtain at each iteration is not strictly increasing, in which case the algorithm may not finish executing. Therefore, in Line 14 we check if the new ε is smaller than the previous one, and if it is, we multiply the previous ε by a value $\xi > 1$ and return that value as the new ε.

Lines 18 through 26 present the near-join trajectory filter algorithm pseudo-code. In Lines 19 and 20 we rasterize the *P* and the *Q* trajectories by placing a uniform grid *G* over the space in which the trajectories move. This is done by splitting every $p \in P$ into a set of sub-trajectories or tracks called *Tracks*(*p*), and every $q \in Q$ into a subset of sub-trajectories called *Tracks*(*q*). Let's call $Tracks(P) \coloneqq \cup_{p \in P} Tracks(p)$ and $Tracks(Q) \coloneqq \cup_{q \in Q} Tracks(q)$. For each track in *Tracks*(*P*) we consider its extended Minimum Bounded Rectangle (eMBR), which is a regular MBR that has been expanded by ε in the horizontal (both to the east and west) and vertical (both to the north and south) directions. Similarly, for every track in *Tracks*(*Q*) we consider its regular MBR (which is an eMBR with ε = 0). The reason why we choose ε = 0 for the eMBRs of the *Q* trajectory set is because we make the assumption that the database (*Q*) is larger than the

query set (P), so that by keeping the eMBRs of Q trajectories small we can ensure that the arrays generated in Line 21 (Figure 19) will be small. Then, we generate a pair (p_t, c) for every grid cell $c \in G$ that trajectory track p_t's eMBR intersects, and a pair (q_t, c) for every grid cell $c \in G$ that trajectory track q_t's MBR intersects. Then, in Line 21 we generate the set of all pairs of tracks (p_t, q_t) such that the eMBR of p_t and the MBR of q_t both intersect the same cell. For example, assume for now that $P = \{1\}$, $Q = \{7,9\}$, that a grid G is given and, without loss of generality, that each trajectory in P and Q only has a single track. Also assume that trajectory 1's track $p_1$has an eMBR(ε) intersecting grid cells c_1 and c_2, and trajectory 7's track p_7 has an eMBR(0) intersecting grid cell c_2 and c_3, and trajectory 9's track p_9 has an MBR intersecting grid cell c_4. Then, NEAR-JOIN FILTER(P,Q,0.1,G) will return {(1,7)} because the eMBRs of both trajectories 1 and 7 intersect with a grid cell in G.

In Line 23 the algorithm finds for all pairs of tracks (p_t, q_t) the identifiers of both p_t and q_t. In the end of the near-join trajectory filter algorithm, we obtain for every $p \in P$ a set of pairs $C_p \subseteq \{(p, q) \in P \times Q\}$, such that for every $(p, q) \in C_p$ it is the case that q is a candidate trajectory that may be within the top K Q-trajectories most similar to p (according to the Hausdorff distance).

Lines 43 through 51 describe the refine stage of our proposed algorithm. In Line 45 we find the set C_p of all the candidate pairs (p,q) associated with trajectory p, and calculate the exact Hausdorff distance between p and q. Then we sort all the elements of $(p, q) \in C_p$ by increasing the Hausdorff distance between p and q, and take the first K elements

corresponding to the closest Q-trajectories to p. For example, if $K = 1$, $P = \{1,3,4\}$, $Q = \{6,7,9\}$ and $C_1 = \{((1,6,1.7), (1,7,1.2), (1,9,2.1))\}$, then we sort C_1 according to the third component of the pairs inside C_1 and obtain $C_1 = \{((1,7,1.2), (1,6,1.7), (1,9,2.1))\}$. Then, since $K = 1$, we take the first element from C_1 as the result. We will follow this procedure for every $i \in P$.

2.4 *Parallel query execution of top-K trajectory similarity queries on GPUs*

In this section we explain how we store trajectories and how we implement each of the functions in our algorithm, shown in Figure 19, on a GPU.

To store the trajectories we follow U2STRA's approach [ZYG12], which we now describe. Each trajectory is divided into disjoint sub-trajectories called tracks. Each track consists of a time-consecutive set of points. We keep three arrays in global memory: the track index array (TKI), the point index array (PTI), and the array of points. Each entry in the TKI array contains information for a single trajectory, so that *TKI*[*j*] is the index in the PTI array of the first trajectory track belonging to the *j*th trajectory. Each entry in the *PTI* array contains information for a single trajectory track, so that *PTI*[*k*] is the index in the array of points of the Kth trajectory. This approach has the advantage that the points belonging to any track are arranged in consecutive memory locations, which facilitates coalesced global memory accesses when loading points of tracks into the thread blocks.

For implementing the function ESTIMATE_ε on a GPU, we assign a thread block B_p for every p ∈ P_sample. Each thread in a given thread block B_p is then in charge of

computing *hausd*(p,q) for a single $q \in Q$. This helps with load balancing because all threads process the same number of trajectories, and all trajectories have approximately the same length. Then, for every thread block B_p, the threads inside B_p will collaborate in sorting the list S_p (see Line 10) in increasing order of Hausdorff distance. Once this is done, a single thread inside each thread block B_p will take the Kth smallest element in the sorted list S_p and write it to array *KNearest*, which resides in global memory. This operation introduces thread divergence, but only within a single warp, so the performance penalty is not big. Then, by performing a parallel prefix sum (see [Ble90], [HSO07]) we can add all the elements in *kNearest* and then divide the sum by $|P_sample|$. In this way, we obtain the average Hausdorff to the K-th nearest Q-trajectory.

For implementing the function TOP-K TRAJECTORY SIMILARITY (Figure 19) on a GPU, we find, in Line 31, the set $D = \{(p, cnt) \mid p \in P, cnt = |\{(p,q) \in PQ_candidates | q \in Q\}|\}$ as follows. First, we take the set PQ_candidates⊆P×Q and sort it in parallel using the first component (the P component) as key. Then we perform a run-length encoding (RLE), which can be efficiently parallelized on GPUs [FHL10], and whose output is the list D. In Line 35, to find the set *Incomplete* consisting of all those trajectories which do not have at least K candidates, we assign to the ith element $(p, count)_i \in D$ a thread t_i, and this thread will output *isIncomplete*[i] if $count < K$, or a 0 if not. Once we have the *isIncomplete* array, we can perform a parallel exclusive sum over *isIncomplete* and obtain an *offsets* array. This achieves load balancing because all threads in all blocks perform the same amount of work. Then, thread t_i takes the

isIncomplete[i], the *offsets*[i] and the D array entries, and performs the instruction *Incomplete*[*offsets*[i]]←D[i] if *isIncomplete*[i] = 1.

For implementing the function REFINE_TKSIMGPU (Figure 19) on a GPU, in Line 44 we assign an element p to every thread block B_p, so that B_p will be in charge of the subset $C_p = \{(p, q) \in PQ_candidates\}$ associated with p. Here there is the potential that there might be some load imbalance because different $C_p's$ might have different cardinalities. However, with our epsilon estimation algorithm (based on sampling) we can expect that the cardinalities of different $C_p's$ will have a small variance and the load imbalance will be tolerable. To find C_p for every p we perform a run-length encoding on the array $\prod_P PQ_candidates$ (the projection on the left component of the tuples in *PQ_candidates*) and we obtain two arrays *unique* and *counts*. Then, by doing an exclusive parallel sum over the *counts* array we can obtain the offsets on the *PQ_candidates* array at which the blocks need to start reading their assigned elements. After this, each thread t inside thread block B_p will be in charge of finding the Hausdorff distance between a different pair $(p, q_t) \in C_p$. Once this is done, the threads inside thread block B_p will sort their assigned elements, in parallel, in increasing order of their Hausdorff distances, and then the smallest K elements ($Result_p$) are written to the global memory.

The GPU implementation of the function NEAR_JOIN_FILTER (Figure 19) follows [ZYG12], and is now explained. In this discussion, we assume that every track in $Tracks(P) \cup Tracks(Q)$ has a *unique* identifier. We generate four arrays of integers:

VQQ, *VQC*, *VPP*, and *VPC*. These arrays satisfy the following properties. Both *VQQ* and *VQC* have the same length, and both *VPP* and *VPC* have the same length, which may be different from the length of *VQQ*. If the eMBR of a track with identifier *j* belonging to a trajectory in *P* intersects with grid cell *k*, then there will be a non-negative integer *m* such that $VQQ[m] = j$ and $VQC[m] = k$. In a similar way, if the eMBR of a track with identifier *j* belonging to a trajectory in *Q* intersects with grid cell *k*, then there exists a non-negative integer *n* such that $VPP[n] = j$ and $VPC[n] = k$. Once these four arrays have been constructed, the technique will sort the arrays *VQQ* and *VQC* using *VQC* as keys, so that many consecutive entries in the *VQQ* arrays correspond to the same grid cell. The idea is now to generate a candidate set of pairs of trajectory tracks $C \subseteq Tracks(P) \times Tracks(Q)$, such that $(VQQ[k], VPP[l]) \in C$ if and only if $VPC[k] = VQC[l]$. In other words, if the eMBRs of two trajectory tracks intersect the same cell, then the pair consisting of those two tracks will belong to the candidate set. This set is found by assigning to every entry $VPP[k]$ a thread, and this thread will then perform a binary search on the array *VQC* to find the smallest integer *l* such that $VQC[l] = VPC[k]$ and the largest integer *m* such that $VQC[m] = VPC[k]$. When processing top-K trajectory queries with our TKSimGPU algorithm, we observe, based on our experiments, that the best performance is not achieved with larger grid sizes (grids larger than 512×512). The reason for this is that with those large grid sizes, the function NEAR_JOIN_FILTER tends to generate very large arrays in Line 21 of Figure 19 because any given trajectory will then intersect many of these small grid cells. This is particularly problematic on GPUs because of their small global memory size, and because we also need to keep the database of trajectories in main memory. To solve this issue one can pick a smaller grid

size (in our experiments we chose the grid size 128×128). Another approach one could take to address the memory issue is to split the query set size (*P*) into subsets, and make separate calls to TKSimGPU.

3 The Top-KaBT Algorithm: A GPU Technique for Pruning Candidate Sets that Arise when Processing Top-K Trajectory Queries

3.1 *Motivation of Top-KaBT*

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.

Although TKSimGPU has been shown to work well with small data sets, it still generates many spurious candidate trajectory pairs that carry an associated performance penalty. For this reason, we introduced Top-KaBT, a GPU technique to reduce the number of spurious candidate trajectory pairs generated by Top-K trajectory similarity query algorithms for Big Trajectory Data applications, and help diminish the negative

impacts that spurious candidate trajectory pairs have on the overall performance of top-K trajectory similarity query processing algorithms.

3.2 *Overview of Top-KaBT*

Top-KaBT is a parallel GPU algorithm for reducing the number of spurious candidate trajectory pairs $(p, q) \in P \times Q$ generated by top-K trajectory similarity query GPU algorithms that follow the filter-and-refine schema and use a trajectory similarity that satisfies the properties of a metric. An example of such a parallel algorithm is TKSimGPU [LGZY15]. The relationship between Top-KaBT and the underlying trajectory similarity query algorithm is illustrated in Figure 20. This figure shows that the similarity query processing algorithm's filter stage generates a set of pairs *(p,q)* that is cheaply pruned by Top-KaBT, and then the output of Top-KaBT is fed back into the similarity query processing algorithm's refine stage in order to produce the query result. To accomplish its goal, Top-KaBT calculates lower and upper bounds of the Hausdorff distance between p and q for every candidate pair $(p, q) \in P \times Q$. These calculations are much cheaper than the calculations of the Hausdorff distances (as shown in Figure 20), a fact that will be proved in Section III.3.3. 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 GPU'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.

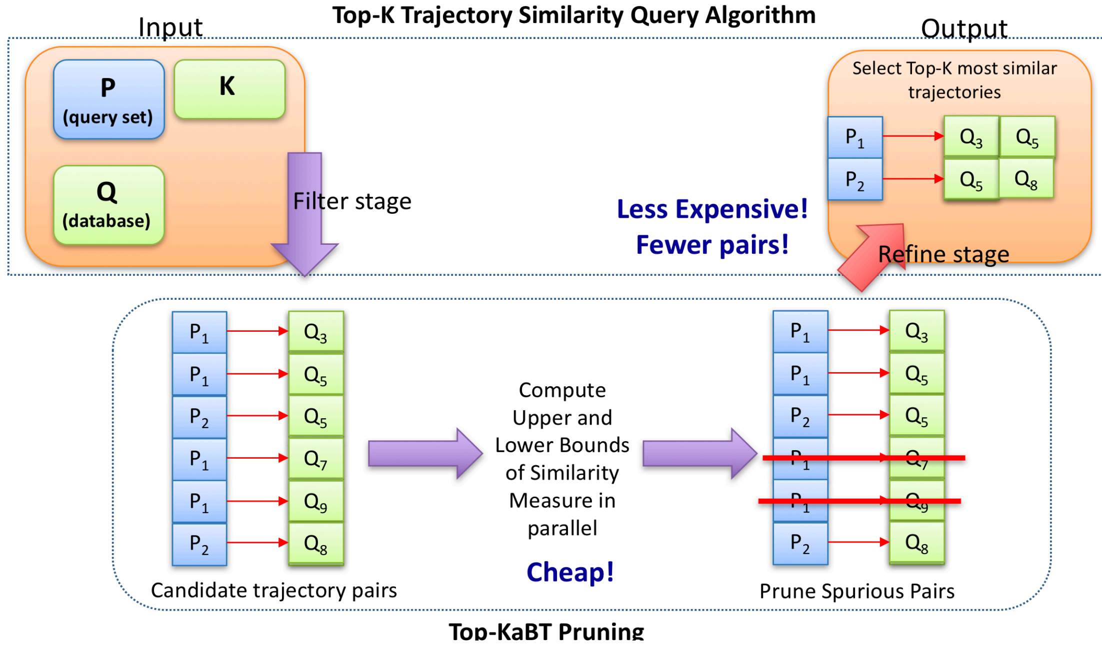

Figure 20. Relationship between Top-KaBT and top-K trajectory similarity query processing algorithms

3.3 *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 $C_p = \{(p, q_0),\ (p, q_2), \cdots, (p, q_{n_p-1})\}$ sorted by the lower bounds to their respective Hausdorff distances, then if we find an integer v_K such that $0 \leq v_K \leq n_p - 1$, and v_K meets certain conditions explained later, we will know that the K most similar trajectories to p will be among $C_p = \{(p, q_0),\ (p, q_2), \cdots, (p, q_{v_K})\}$, and we can prune the remaining elements $C_p = \{(p, q_{v_K+1}),\ (p, q_{v_K+2}), \cdots, (p, q_{n_p-1})\}$.

In the remainder of this section, we use m_{p,q_i} to refer to the value $\min_{x \in MBR(p), y \in MBR(q)} d(x,y)$, 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 $\max_{x \in MBR(p), y \in MBR(q)} d(x,y)$.

Lemma 3.3.1. For any $(p,q) \in P \times Q$, it is true 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 3.3.2 (Cut point set). Given the candidate set $C_p = \{(p,q_0),(p,q_1),(p,q_2),(p,q_3)\}$ 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*.

Example 3.3.3. 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 $CP_p = \{1,2\}$ 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 3.3.4 (Min-cut point). Given the candidate set $C_p = \{(p,q_0), (p,q_1), \cdots, (p,q_{n_p-1})\}$ satisfying $m_{p,q_i} \leq m_{p,q_{i+1}}$ 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 $min\, CP_p$.

Example 3.3.5. In Example 3.3.3 the min-cut point is $min\, CP_p = 1$.

Definition 3.3.6 (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 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 CP_p.

Example 3.3.7. In Example 3.3.3 the min-K-cut point for $K = 2$ is 2.

Theorem 3.3.8. If v is a cut point of the following candidate set $C_p = \{(p, q_0),\ (p, q_1),\ \cdots, (p, q_{n_p-1})\}$ 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 the following inequality holds $hausd(p, q_v) \leq M_{p,q_v} \leq hausd(p, q_{v+1})$. 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 3.3.9. If v_K is a min-K-cut point of the candidate set $C_p = \{(p, q_0),\ (p, q_1),\ \cdots, (p, q_{n_p-1})\}$, 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 $\{q_i | 1 \leq i \leq v_k\}$. 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 $\{q_i \, | 0 \leq i \leq v_{k+1}\}$, which is what we wanted to prove.

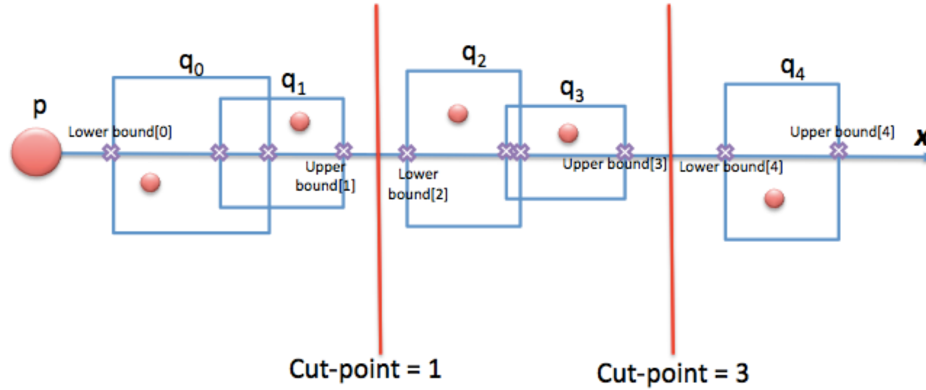

Figure 21. Example of K-cut point

Example 3.3.10. Continuing with Example 3.3.3 and using Theorem 3.3.9, 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 3.3.11. In Figure 21 we have an object p and five objects q_0, q_1, q_2, q_3, and 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 *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 *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 that *LowerBound[i]* ≤ *LowerBound[i+1]*, so that the antecedent of Theorem 3.3.9 holds in this case. Additionally, it can be seen that *UpperBound[1]* ≤ *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 is smaller than the closest that q_2 could be 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 3.3.9, 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 *UpperBound[3]* ≤ *LowerBound[4]*, which means that 3 is a 2-cut point of the candidate set in the figure. Therefore, according to Theorem 3.3.9, 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 3.3.12. The minimum Euclidean distance between two MBRs R with the lower-left corner (r_x,r_y) and the upper left corner (r'_x, r'_y), and S with the lower-left corner (s_x,s_y) and the upper left corner (s'_x, s'_y), can be computed in constant time complexity using the mindist formula of [RRS00]): $mindist(R,S) = \sqrt{d_x^2 + d_y^2}$, 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 3.3.13. Given a candidate set $C_p = \{(p, q_0),\ (p, q_1),\ \cdots, (p, q_{n_p-1})\}$, Observation 3.3.12 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 $(p, q_i) \in C_p$.

3.4 *Description of Top-KaBT's Pruning Strategy*

In this subsection we describe our proposed parallel GPU technique to prune the candidate set of the top-K trajectory similarity query processing algorithm, which is based on Theorem 3.3.9. The pseudocode algorithm for this technique is in Figure 22, while Figure 23**,** Figure 24**,** Figure 25, and Figure 26 provide an illustrated example.

The main function is called SORT_PRUNING and is presented in Line 1 of Figure 22. 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 3.3.9. 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 (low_p and up_p, respectively) of the trajectory similarity between *p* and *q*, using Observation 3.3.13.

Algorithm Prunes top-K trajectory similarity candidate set

Input: An integer $K > 0$. A set of pairs $candidates \subseteq P \times Q$.

Output: A set $Result \subseteq candidates \subseteq P \times Q$ that contains the result of a top-K trajectory similarity query, when processed with TKSimGPU.

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: **function** REMOVE($candidates, cut_pts$)
36:     $n \leftarrow |\Pi_P(candidates)|$
37:     **for** $i \in \{0, 1, \ldots, n - 1\}$ **do in parallel**
38:         $offset[i] \leftarrow |\{(p_j, q) \in candidates \mid p_j < p_i\}|$
39:         $B[2i] \leftarrow cut_pts[i] + offset[i] + 1$
40:         $B[2i + 1] \leftarrow |\Pi_P(candidates)[i]|$
41:         $Alter_1s0s[2i] \leftarrow 1$
42:         $Alter_1s0s[2i + 1] \leftarrow 0$
43:     **end for**
44:     $Counts \leftarrow adjacentDifference(B)$
45:     $Stencil \leftarrow RLD(Counts, Alter_1s0s)$
46:     $PfxStencil \leftarrow ExclPfxSum(Stencil)$
47:     **for** $i \in \{0, \ldots, n - 1\}$ **do in parallel**
48:         **if** $Stencil[i] = 1$ **then**
49:             $Pruned[PfxStencil[i]] \leftarrow candidates[i]$
50:         **end if**
51:     **end for**
52:     **return** $pruned$
53: **end function**
54: **return** SORT_PRUNING($K, candidates$)

Figure 22. Sort pruning algorithm of Top-KaBT

This is illustrated in Step 1 in Figure 23, 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, low_p, and up_p, using the entries in low_p as keys; in this way we ensure that the premise of Theorem 3.3.9 is satisfied. An example of this is shown in Step 2 in Figure 23, 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_3) has smaller lower bound (whose value is 1.3) than (p_3, q_7), which has 2.7 as a lower bound,

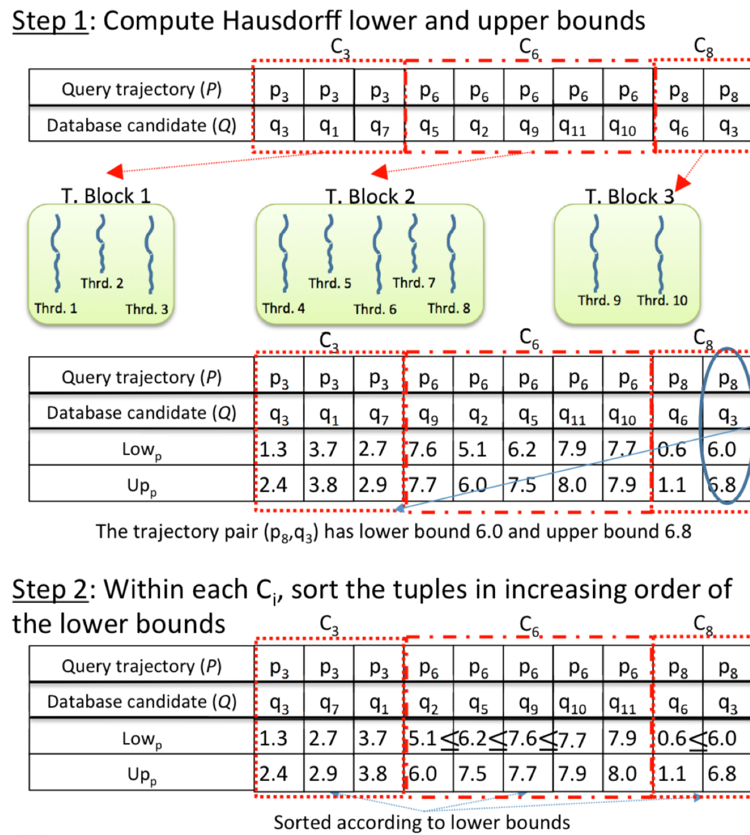

Figure 23. Example run of the sort pruning algorithm of Top-KaBT (Steps 1 and 2)

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 22, low_p is shifted 1 entry to the left for memory coalescing in line 23. The reason for this is that, according to Theorem 3.3.9, 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 $m_{p,q0}$ is never used. Figure 25 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 3.3.9, 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 24 and Figure 25, respectively.

The function HAUSDORFF_BOUNDS in Line 15, shown in Step 1 in Figure 22, 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 3.3.13. 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 in Figure 22 receives as input parameters a p in P, an integer K, 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 3.3.9. After the parallel loop in Lines 24 through 30, an array cut_pt_p is obtained, which is shown in Step 4 in Figure 24. 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 inclusive prefix sum [HSO07] over cut_pt_p (which is the portion of the *cut_pt* array corresponding to p) is performed to obtain the array *Pfx_cut_pt* of Line 31; this is shown in Step 4 in Figure 24 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

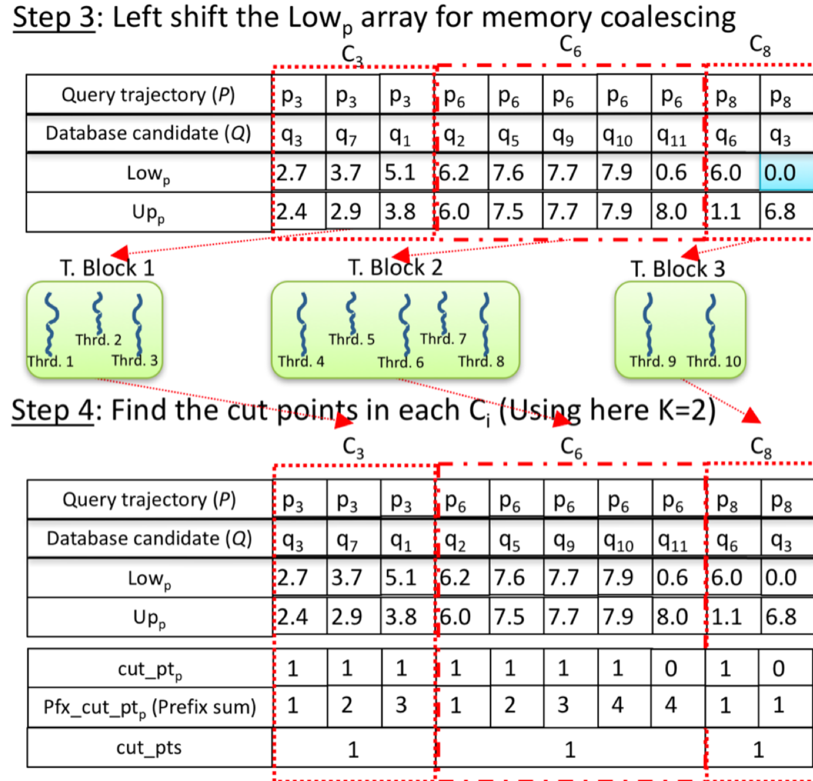

Figure 24. Example run of the sort pruning algorithm of Top-KaBT (Steps 3 and 4)

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 does 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.

The function REMOVE in Line 35 in Figure 22 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*). This last array satisfies that *cut_pts[i]* is the cut point

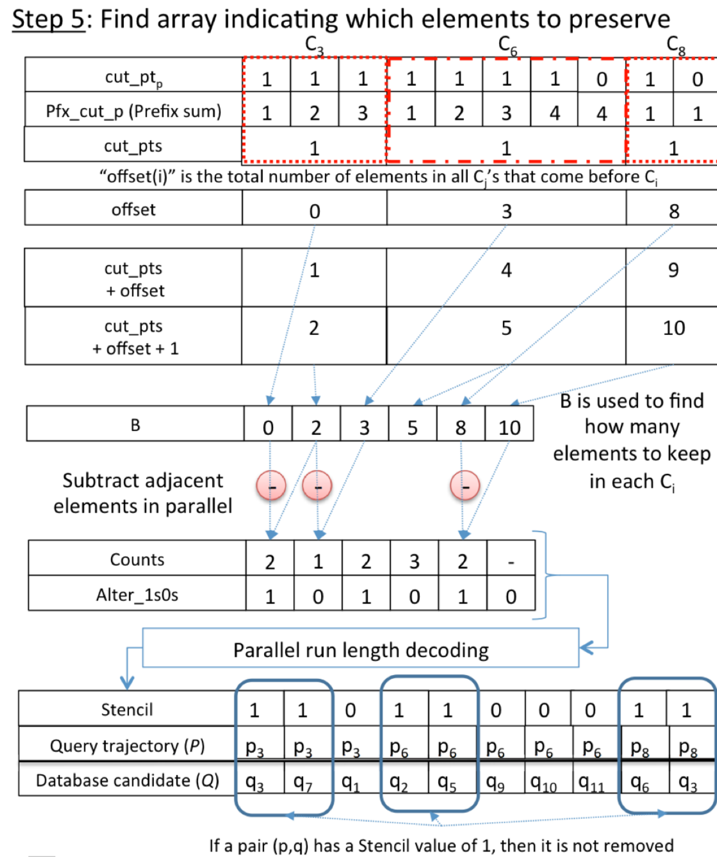

Figure 25. Example run of the sort pruning algorithm of Top-KaBT (Step 5)

associated with the i-th p trajectory in $\Pi_p(candidates)$. Lines 37 through 43 create an array *B* that contains the elements of *cut_pts + offset + 1* in its even-indexed entries, and the elements of $|\{(p_j, q) \in candidates | p_j < p_i\}|$ in its odd-indexed entries. In Figure 25 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)* are going to be preserved for every p. In these same Lines 37 to 43, we 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 [FHL10]. Then Line 44 performs a parallel reduction to compute the array *Counts* satisfying that *Counts[i] = B[2i+1]-B[2i]*. *Counts[2*i+1]* is the number of q candidates associated with p_i that can be pruned away, while *Counts[2*i]* indicates the number of q candidates associated with p_i that cannot be pruned away. In Figure 25 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 p_3 (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_1s0s* (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 3.3.9.

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 Step 6 in Figure 26 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.

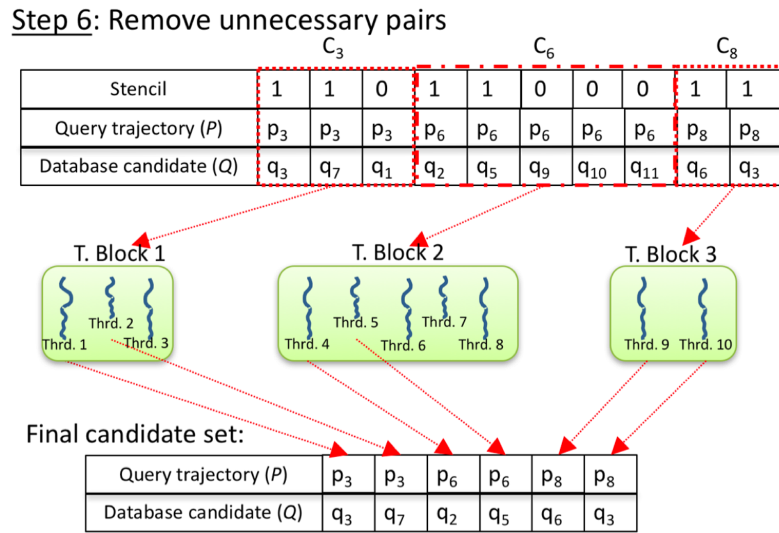

Figure 26. Example run of the sort pruning algorithm of Top-KaBT (Step 6)

4 The TrajEstU Algorithm: A GPU Technique for Reducing Trajectory Uncertainty when Processing Top-K Trajectory Queries

4.1 *Motivation of TrajEstU*

As we have mentioned before in Section I.3, trajectories have an associated error stemming from the noise of location sensors (measurement error), and from the fact that trajectories are approximations to the true paths of the objects (model error). These errors can introduce significant deviations in the results of top-K trajectory similarity queries. To address this problem, we have proposed an algorithm called *TrajEstU*, which estimates the true path of the objects, and then generates a new estimated trajectory. The idea is then to run TrajEstU as a preprocessing step (i.e. before running TKSimGPU + Top-KaBT) over the database of trajectories (*Q*), and also run it on the query set (*P*) of the top-K trajectory similarity queries; thereby reducing the negative impacts of both kinds of errors when processing this type of trajectory queries.

4.2 *Overview of TrajEstU*

TrajEstU receives as input a trajectory database *db* and an uncertain input trajectory *p*. To estimate the true path of the trajectory *p* of a moving object in an unconstrained space when there is uncertainty due to measurement errors and/or low sampling rates, TrajEstU goes through three stages: (i) a pre-processing stage, (ii) a model fitting stage, and (iii) a trajectory generation stage.

In the first stage, the pre-processing stage, TrajEstU performs local segment clustering of the trajectories in *db* [LHW07] with the intention of finding the spatial patterns that the trajectories in the database exhibit. The output of this local segment clustering

algorithm is a set of segment clusters, each of which has an associated representative trajectory describing the behavior of its cluster. Once the database trajectories have been locally clustered, TrajEstU builds an R-tree *clusterTree* containing the representative trajectories of each of the segment clusters found.

In the second stage, the model fitting stage, TrajEstU identifies the time intervals where p has a near-constant acceleration. Then, for each one of these time intervals I, the algorithm finds the extended Minimum Bounding Rectangle (eMBR) containing the points of p with timestamps contained in I (this set of points is denoted by $p[I]$) and uses this eMBR to perform a range search over *clusterTree*. The output of this range search is a set of representative trajectories called *clustersI*. Then, for each representative trajectory r in *clustersI*, TrajEstU builds a separate linear constant acceleration model for $p[I] \cup r$. The result of this operation is one candidate constant acceleration model for $p[I]$ per representative trajectory r. Out of all these models for $p[I]$, TrajEstU chooses the one with the highest goodness of fit. The collection of constant acceleration models for all intervals I makes up the kinematic model for p, and allows us to predict the true path of the object at any given time.

In the final stage, given the kinematic model found in the second stage, TrajEstU generates a new trajectory with uniform sampling rate, called the *estimated trajectory*, whose points are the ones predicted by the kinematic model.

4.3 *Description of TrajEstU*

4.3.1 Pre-processing stage

In this section we explain the first phase of TrajEstU, which consists of the preprocessing operations that need to be performed only once. Then, *as more query trajectories arrive, these operations need not be performed again*. In this stage, the idea is to identify the local patterns displayed by the set of moving objects in *db*. To this end, we perform local trajectory clustering [LLKH10] of the trajectories in the database, and find the representative trajectory associated with each cluster. It is in these local cluster trajectory representatives that the movement patterns are condensed.

Our proposed algorithm uses local trajectory clustering to find local trajectory patterns. Lee et al. [LHW07][LLLH10] introduced the idea of first partitioning a set of trajectories into segments and then clustering the resulting segments, instead of clustering the trajectories as a whole. This serves our objectives because by clustering trajectories into segments, we can obtain the movement patterns in a given small area, instead of globally clustering the trajectories, which would not be able to discover patterns at a local scale.

4.3.2 Model-Fitting Stage

In this section we explain TrajEstU's second phase, the model-fitting stage. We describe the kinematic trajectory model used and how to estimate its parameters. The model is based on kinematics. First, we identify the time intervals of the object's trajectory where it has constant acceleration, and build a constant acceleration model for each of these intervals. The collection of these constant acceleration models makes up the kinematic trajectory model. Then, the constant acceleration models for any two

consecutive time intervals $[t_0, t_1]$ and $[t_1, t_2]$ need to be smoothly connected in order for them to be consistent around t_1.

4.3.2.1 Constant Acceleration Model

It is known that if an object o with initial position x_0 and initial velocity v_0 moves *with constant acceleration* a during a time interval $[t_0, t_0 + \Delta_t]$, $\Delta_t > 0$, we can then accurately determine the position of o at any time $t_0 + t$, with $0 < t < \Delta_t$ using $\mathrm{x(t)} = \mathrm{x0} + \mathrm{v0} \cdot \mathrm{t} + \mathrm{a} \cdot \mathrm{t}^2/2$. This is the time-linear dynamic trajectory model that we will use for our trajectories, and its parameters are x_0, v_0 and a.

In the case of an uncertain trajectory, the problem is that it consists of a sequence of only *uncertain* positions (called sampled points or observations), so the velocity and acceleration, if computed straightforwardly from the observed positions, are also uncertain quantities. To address this problem, we find the best linear constant acceleration trajectory model that fits the observed positions by using a standard linear regression model [LLD06]. The form of the linear model is $Z = HX$, where Z is the observation vector, H is the model matrix, and X is the parameter vector. As is the case in standard linear regression, we seek to find the parameter vector X that minimizes the sum of the squares of the errors $(Z - HX)^T(Z - HX)$. We now explain with an example how Z, H and X are found.

Suppose for example that we are given the sequence of $n = 3$ points $\{(x_0,y_0,t_0), (x_1,y_1,t_1), (x_2,y_2,t_2)=(x_{n-1},y_{n-1},t_{n-1})\}$ belonging to an uncertain trajectory. To find the best

linear constant acceleration trajectory model fitting these data, we compute $\Delta t_i = t_i - t_{i-1}$ for $1 \leq i < n = 3$, and then build the model matrix H of size $(2n)\times 6 = 6\times 6$ as follows:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & \Delta t_1 & \Delta t_1^2 & 0 & 0 & 0 \\ 1 & \Delta t_2 & \Delta t_2^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t_1 & \Delta t_1^2 \\ 0 & 0 & 0 & 1 & \Delta t_2 & \Delta t_2^2 \end{bmatrix}$$

and the observation vector Z of size $(2n)\times 1 = 6\times 1$ as follows:

$$Z = [x_0 \quad x_1 \quad x_2 \quad y_0 \quad y_1 \quad y_2]^T$$

Then, we use the standard formula $X = (H^T H)^{-1} H^T Z$ to compute the parameter vector corresponding to the linear constant acceleration trajectory model that minimizes the sum of the squares of the residuals $(Z - HX)^T (Z - HX)$:

$$X = (H^T H)^{-1} H^T Z = \left[x_0 \quad v_0^{(x)} \quad \frac{1}{2} a^{(x)} \quad y_0 \quad v_0^{(y)} \quad \frac{1}{2} a^{(y)}\right]^T,$$

where the vector (x_0, y_0) is the initial position of the object during the interval *[t_0, t_1]*, $v_0 = (v_0^{(x)}, v_0^{(y)})$ is the initial velocity, and $a = (a^{(x)}, a^{(y)})$ is the constant acceleration during that interval. This means that the fitted model during the interval *[t_0, t_1]* has x-component $m^{(x)}(t) = x_0 + v_0^{(x)} t + 0.5 a^{(x)} t^2$, and y-component $m^{(y)}(t) = y_0 + v_0^{(y)} t + 0.5 a^{(y)} t^2$.

One key assumption made when building the above linear model, besides that the acceleration is constant, is that the trajectory has a high-sampling rate during the time interval corresponding to the observed data, which means that there are enough data from which to discover the parameters of the model. However, when building a linear

model for low-sampling rate trajectories, there may not be enough data to build an *accurate* model. To solve this problem, we can exploit the knowledge provided by the database of trajectories by mining the movement patterns described by the trajectories in the database around the spatial area where our given trajectory query has a low-sampling rate. To discover these movement patterns we use a trajectory clustering algorithm. However, algorithms for clustering trajectories may not be appropriate for discovering the movement patterns around a particular area of interest (where the given trajectory has a low-sampling rate) because regular trajectory clustering algorithms cluster trajectories globally. Instead, we propose using an algorithm that performs local clustering of trajectory segments [LHW07] because it first splits trajectories into smaller segments and then clusters the segments. To each cluster, the clustering algorithm assigns a representative trajectory that captures the behavior of the segments in the corresponding cluster. By incorporating the knowledge of these representative trajectories into the constant acceleration model, we can overcome the difficulty of building a linear model for low-sampling rate sections of trajectories.

4.3.2.2 Incorporation of Trajectory Patterns

Once the database trajectories have been locally clustered, we have the knowledge of the spatial patterns that the trajectories in *db* exhibit in space. To exploit this knowledge, during the generation of a constant acceleration model in the time interval $I=[t_0, t_1]$ with $t_0 < t_1$, we do the following. We first construct an extended minimum bounding rectangle eMBR(ε) with $\varepsilon > 0$ surrounding the sampled points of the trajectory with times in the range $I = [t_0, t_1]$. Then, we locate the set of clusters that

intersect with this eMBR. This is illustrated in Figure 27, where we see the input trajectory p. In that figure we have that $t_0 = p[2].t$ and $t_1 = p[3].t$. There we see that the intersection of the eMBR(ε) of $p[p[2].t, p[3].t]$ (the set of points of trajectory p with timestamps within the interval $[p[2].t, p[3].t]$) with *Clusters* is the list $Cluster_I$ = *{Cluster1, Cluster2, Cluster3}*. We will consider each of these clusters individually, and for each one, we will build a constant acceleration model using the points in the set $p[t_0, t_1] \cup Clusters_I[i]$, where $Clusters_I[i]$, denotes the ith cluster in the list $Clusters_I$. However, one obstacle here is that the points in $Clusters_I[i]$, unlike the points in $p[t_0, t_1]$, do not have timestamps; therefore, we cannot directly build the matrix H, which depends on the timestamps of the points.

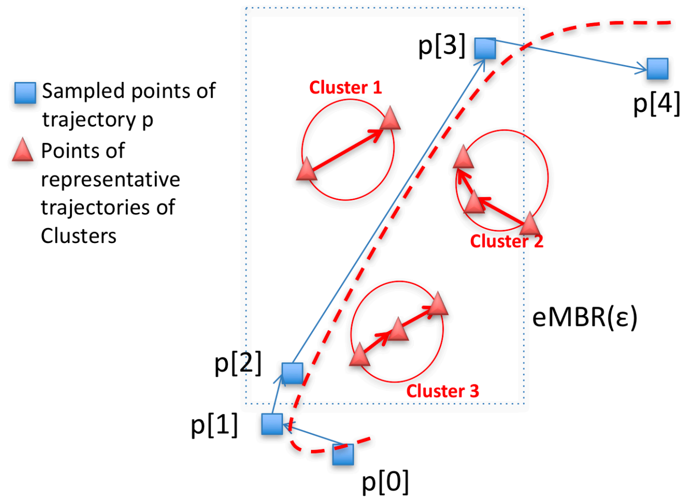

Figure 27. Finding representative trajectories

To solve this problem, we assign times to these points using the closest trajectory points. To this end, we associate an empty list L_k with every point *p[k]* in the trajectory p. Then for every point c_j in *Clusters[i].repr* (the representative trajectory associated with the i-th cluster in the list *Clusters*) we find the closest consecutive pair of trajectory points *p[m]*, *p[m+1]* and insert c_j into L_m. This is illustrated in Figure 28 where we see

that the points c_0, c_1 and c_2 in *Cluster3* have the consecutive pair of trajectory points p_2 and p_3 as the closest one; so $L_2=[c_0, c_1, c_2]$. Then we consider the list of points [*p[2]*, c_0, c_1, c_2, *p[3]*] and use this list to linearly interpolate the timestamps for c_0, c_1 *and* c_2, assuming a uniform sampling rate. Therefore, the timestamp of c_1 will be $\frac{2}{5}(p[3].t - p[2].t)$, and the timestamp for *p[2]* will *be* $\frac{3}{5}(p[3].t - p[2].t)$. Once every point c_j in $Clusters_I[i]$ has been assigned a timestamp, we can proceed with the fitting of a candidate constant acceleration model for the set of points $p[t_0, t_1] \cup Clusters_I[i]$.

4.3.2.3 Selecting the Best Constant Acceleration Model

In Subsection III.4.3.2.2 we examined how we can build a constant acceleration model for a trajectory *p* during the time interval $[t_0, t_1]$ with $t_0 < t_1$, using a single segment cluster in the database found by intersecting the eMBR(ε) surrounding the points of $p[t_0, t_1]$ with the set of segment clusters *Clusters*. However, the result of this

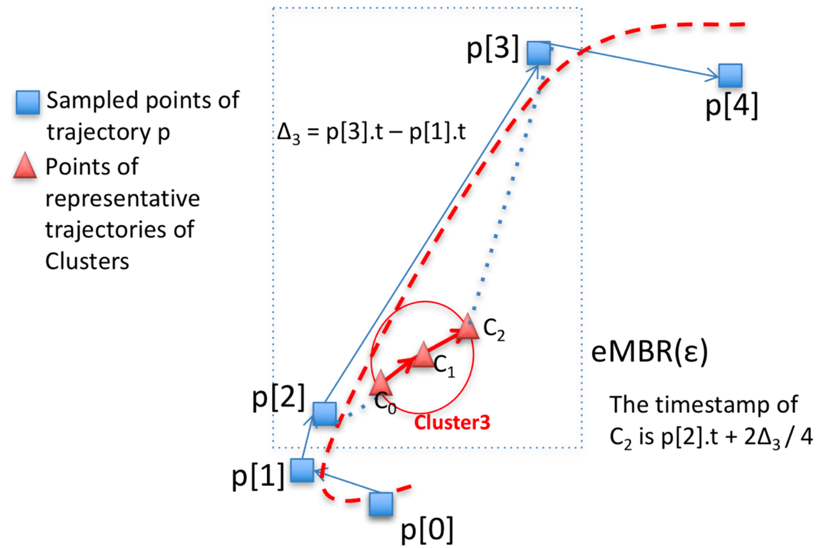

Figure 28. Timestamp calculation for a set of cluster points

intersection could yield more than one intersecting segment cluster, which would imply that we have one candidate constant acceleration model per intersecting cluster. Out of all these possible constant acceleration models for $p[t_0, t_1]$, we pick the model that maximizes the goodness of fit, i.e., the one that best explains the variance of the data, using the standard formula for the coefficient of determination.

4.3.2.4 Variable Acceleration Model

A moving object could, however, change its acceleration throughout the extent of its movement, rendering the above constant acceleration model incapable of accurately predicting the movement of the object. Hence, we identify the time instants at which the moving object changes its acceleration. To accomplish this, we use a real positive number *tol* > *0* as a constant parameter, and keep an integer *startInterval* with the initial value 0, and scan the query trajectory q from the beginning to the end, and computing the average acceleration at each point with an index *i,* starting from *0*. If the absolute value of the difference between the acceleration at *p[startInterval]* and at *p[i]* exceeds the fixed parameter *tol*, then we consider that *[p[startInterval].t, p[i].t]* is a constant acceleration interval. Then we assign i to *startInterval* and then keep scanning the trajectory in search for the next interval.

The average acceleration is, by virtue of being computed from the observed positions of an uncertain trajectory, an uncertain quantity. By using the *tol* parameter as a threshold to identify the time intervals where the trajectory has constant acceleration, we are able to address this uncertainty problem.

4.3.2.5 Model Coupling Function

As discussed in Sections III.4.3.2.1 to III.4.3.2.4, to model the movement of an object, we identify the time intervals where the object's acceleration does not vary beyond a pre-specified tolerance $tol > 0$, and then fit a constant acceleration model for each of these intervals. However, at the time instant that lies at the boundary between two consecutive time intervals, for example, if two intervals of near-constant acceleration are $[t_0,t_1]$ and $[t_1,t_2]$, with $t_0 < t_1 < t_2$, then at time t_1 we have two estimates for the position of the object: one estimate arises from the constant acceleration model during $[t_0,t_1]$, and the other from the model during $[t_1,t_3]$, and these two estimates could potentially differ. To overcome these difficulties we smooth the trajectory model by smoothly connecting, or coupling [GM14], the constant acceleration model during $[t_0,t_1]$ with the constant acceleration model during $[t_1,t_2]$. This is illustrated in Figure 29 where we see two constant acceleration models: Model *p[0].t* to *p[2]*.t, and Model *p[2].t* to *p[4].t*, shown as the pointed and dashed lines, respectively. These models disagree in their estimations around p_2, but when we incorporate the model coupling function, then both models are smoothly connected.

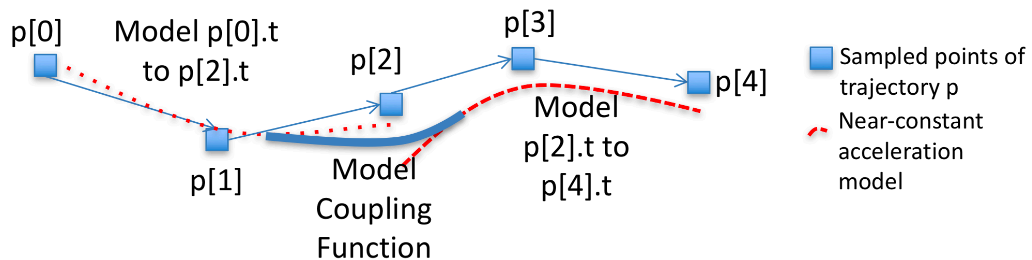

Figure 29. Model coupling function

Assume we have a constant acceleration model m_1 during $[t_0,t_1]$ with $m_1^{(x)}(t)$ and $m_1^{(y)}(t)$ as the x and y-components, respectively, and another constant acceleration model during $[t_1,t_2]$ with $m_2^{(x)}(t)$ and $m_2^{(x)}(t)$ as its x and y-components, respectively. To smoothly connect the two models, we can use the hyperbolic tangent, which is a function that is known to be suitable for this purpose [GM14]. The model coupling function $c(m_1, m_2)^{(x)}(t)$ smoothly connects both models and has an x-component given by $c(m_1, m_2)^{(x)}(t) = m_1^{(x)}(t) + \frac{\tanh(t-t_1)+1}{2}(m_2^{(x)}(t) - m_1^{(x)}(t))$, and the y-component is identical, but replacing x by y. This function smoothly connects both models because *tanh* converges to 1 as t goes to infinity, and to -1 as t goes to minus infinity. Therefore, for large t, $c(m_1, m_2)^{(x)}(t)$ converges to $m_2^{(x)}(t)$, and for t values less than t_1, it converges to $m_1^{(x)}(t)$.

4.3.3 Trajectory Estimation Stage

In this section we explain the final stage of TrajEstU. The purpose of this stage is to generate a trajectory, called the *estimated trajectory*, using the kinematic model found during the model-fitting stage presented in the previous Section 4.3.2. This kinematic model is a collection of constant acceleration models $\{(model_I, I) \mid I$ *is a constant acceleration interval of p*$\}$. Each $model_I$ can be used to estimate the true trajectory of the moving object during the time interval *I*. Given a number of points *numPoints*, and the lifetime *Inter = [startTime, endTime]* of *p*, we can generate a uniform sampling rate trajectory by querying the kinematic model at time instants $\{startTime, startTime + \Delta_t, startTime + 2\Delta_t, ...\}$, thereby obtaining the estimated trajectory.

4.4 *Details of TrajEstU*

In this section we explain the details of TrajEstU, the pseudocode of which is presented in Figure 30. TrajEstU consists of a preprocessing stage where the spatial trajectory patterns of the database are mined, a model-fitting stage where we build a trajectory model, and then a trajectory estimation stage where we use such model to produce an estimated trajectory.

4.4.1 Pre-processing Stage

First, the algorithm invokes the function *cluster_segments* implementing the local clustering algorithm [LHW07] to get the set of segment clusters for all trajectories in the database *db* (Line 44). Once this is done, the algorithm constructs an R-tree [Gutt84] containing the representative trajectories of each local segment cluster (Line 45). If there are multiple query trajectories, then *this stage is to be executed only once off-line, so it does not to be run with each query*.

4.4.2 Model-Fitting Stage

Once the pre-processing stage has finished, TrajEstU proceeds to fit a model for every query trajectory by calling function *FILLDATA* (Line 46). The function *FILLDATA* receives as input arguments a query trajectory *p*, a database of trajectories *db*, an R-tree *clusterTree* with the representative trajectories of the clusters, and a set of clusters *clusters*, and is in charge of fitting a kinematic trajectory model for *p*. It first computes the set *intervals* of time intervals where *p* has near-constant acceleration (Line 3). It

then builds for each constant acceleration interval *I* of *p* a separate model describing the behavior of *p* during *I* (Lines 4 – 15). It achieves this by constructing an extended Minimum Bounding Rectangle eMBR(ε) enclosing the portion of *p* during *I* to account for the uncertainty of *q* during *I* (Line 5). Then, it performs a range search over the R-tree *clusterTree* using the eMBR as input to determine the set $clusters_I$ of all trajectory segment clusters located near *p* during *I* (Line 6). It then finds a candidate constant acceleration model for each trajectory segment cluster in $clusters_I$ by calling the function CONSTACCMODEL and computes the goodness of fit for each model (Lines 7 – 10). Finally, it selects the model with the greatest goodness of fit (Lines 11 – 12).

The function *CONSTACCMODEL* in Line 16 takes a set of trajectory points *points* as its input argument, and finds a linear regression model that fits the trajectory points in *points* following the approach explained in Section III.4.3.2. By using this linear regression approach to fit a constant acceleration trajectory model, we are able to address the issue of measurement uncertainty because linear regression does not force the resulting model to agree with every single sampled point. By considering the representative trajectories of the segment clusters, we are able to address the problem that may arise when trying to fit a model with very few data points available.

4.4.3 Trajectory Estimation Stage

After the model-fitting stage is completed, TrajEstU proceeds to compute the lifetime *Inter* of the query trajectory *p* (Line 47) and then invokes the function *ESTIMATETRAJECTORY* to estimate the true trajectory of *p* (Line 48). The function

ESTIMATETRAJECTORY, shown in Line 30 of Figure 30, receives as input parameter a kinematic model *models* that is a set of pairs of the form ($model_I$, I), where every $model_I$ is a constant acceleration model valid during the time interval *I*. This function also receives as an input parameter the number of points (*numPoints)* of the final trajectory estimate, and an interval *Inter=[startTime, endTime]* that is the lifetime of the estimated trajectory. This function first creates an empty trajectory (Line 31) that will eventually be the estimated trajectory, and then computes the time span between consecutive points, assuming uniform sampling (Line 32). The function then computes the points of the estimated trajectory one by one (Lines 33 – 41). To do so, it finds the model ($model_I$, I) in *models* that is valid at time τ_i (Line 34 – 37), and computes the estimated position at time τ_i using scalar products (Lines 38 – 39). Finally, it adds the estimated position to the estimated trajectory (Line 40).

```
Algorithm  TrajEstU(p, db)
Input: An input trajectory p. A trajectory database db.
Output: A trajectory traj that estimates p.
 1: function FILLDATA(p, clusterTree)
    // Obtains a list of const. acceleration models for p
 2:     models ← ∅
 3:     intervals ← const_acc_intvls(p)
 4:     for I ∈ intervals do
 5:         mbr_I ← EMBR(ε) of p during I
 6:         clusters_I ← rangeSearch(clusterTree, mbr_I)
 7:         for c ∈ clusters_I do
                // c.repr is the repr. trajectory of c
 8:             model_I^c ← CONSTACCMODEL(q[I]) ∪ c.repr)
 9:             r_c^2 ← goodness_of_fit(model_I^c)
10:         end for
11:         model_I ← argmax_{c∈clusters_I} r_c^2
12:         models ← models ∪ {(I, model_I)}
13:     end for
14:     return models
15: end function
16: function CONSTACCMODEL(points)
    // Fits a constant acceleration model to points
17:     Sort points by increasing timestamp values
18:     H ← new matrix of size (2 · |points| × 6)
19:     Z ← new vector of size (2 · |points| × 1)
20:     H[1, :] ← [1, 0, 0, 0, 0, 0] and H[1 + |points|, :] ← [0, 0, 0, 1, 0, 0]
21:     for i ∈ [2, . . . , |points|] do
22:         Δ_i ← points[i].t − points[i − 1].t
23:         H[i, :] ← [1, Δ_i, Δ_i^2, 0, 0, 0]
24:         H[i + |points|, :] ← [0, 0, 0, 1, Δ_i, Δ_i^2]
25:         Z[i] ← points[i].x
26:         Z[i + |points|] ← points[i].y
27:     end for
28:     return (H^T H)^{-1} H^T Z
29: end function
30: function ESTIMATETRAJECTORY(models, numPoints, Inter)
    // Obtains an estimated trajectory from the models
31:     traj ← empty trajectory
32:     Δ_t ← (Inter.endTime − Inter.startTime)/numPoints
33:     for i ∈ [1, . . . , numPoints] do
34:         τ_i = startTime + i · Δ_t
35:         Find (model_I, I) ∈ models such that τ_i ∈ I
36:         t_i ← τ_i − I.start
37:         // model_I is a vector [x_0, v_0^(x), (1/2)a_x, y_0, v_0^(y), (1/2)a_y]^T
38:         p_i^(x) ← model_I · [1, t_i, t_i^2, 0, 0, 0]
39:         p_i^(y) ← model_I · [0, 0, 0, 1, t_i, t_i^2]
40:         Add point (p_i^(x), p_i^(y), τ_i) to traj
41:     end for
42:     return traj
43: end function
44: clusters ← cluster_segments(db) // Pre-processing
45: Create an r-tree clusterTree with {c.repr | c ∈ clusters}
46: model ← FILLDATA(p, db, clusterTree)
47: Inter ← p's lifetime
48: traj ← ESTIMATETRAJECTORY(model, |p|, Inter)
49: return traj
```

Figure 30. TrajEstU pseudocode

5 TraclusGPU: A parallel GPU technique for local clustering of trajectories

5.1 *Motivation of TraclusGPU*

As we have discussed in Section I.3, among the issues of top-K trajectory similarity query processing are the measurement and the model uncertainties. This is because noisy trajectories can significantly affect the accuracy of top-K trajectory similarity queries. To address these issues, we proposed an algorithm called TrajEstU, which was discussed in Section III.4. The idea underlying TrajEstU is that we can estimate the true path of an uncertain trajectory (called the input trajectory) by building linear regression models that assume near-constant acceleration, and that also take into account the patterns followed by trajectories that are located close to the input trajectory. These local trajectory patterns are gathered through the clustering of a database of trajectories, which is done in TrajEstU's off-line pre-processing stage. So, in order for TrajEstU to estimate the true path of an input trajectory, it needs to run an off-line preprocessing stage (this stage is run at most once per database, and should not be run with every input trajectory) that locally clusters the database trajectories using the Traclus Algorithm [LHW07]. Then, after this off-line preprocessing stage is done, TrajEstU can use these clusters to build models with which it can estimate the input trajectory.

The set of experiments performed on TrajEstU, shown in Section IV.2.3, show that the execution time performance of the online component of this algorithm is negligible, and scales well in terms of the length of the query trajectory, and the number of trajectories in the database, among other parameters. However, the off-line pre-processing stage of TrajEstU (which is performed just once per database, and should not be run with each

trajectory), which consists of locally clustering the database trajectories with the serial version of the Traclus algorithm, does not scale with the number of trajectories in the database. Moreover, for the database sizes used in our experiments (see Section IV.2.3), the serial version of the Traclus algorithm took weeks to finish its execution. Therefore, in order for TrajEstU to have practical application for dealing with Big Trajectory Data, it is essential that its offline trajectory clustering stage scales with the large size of the databases, so that this stage produces results in a reasonable amount of time.

As always, one way of dealing with the large volume of Big Trajectory Data is to make use of parallel computing. In particular, GPUs, as has been discussed in Section I.2.4, are a parallel architecture that possesses many advantages. Among these advantages are: GPUs are available in most computers, including mobile devices, desktops, workstations and supercomputers; GPUs are very energy efficient [LM13]; and, GPUs, for certain tasks, can have up to an order of magnitude of higher floating point throughput than the best multicore chip CPUs available [LKCD+10]. All these advantages of GPUs make them a good candidate architecture in which to develop a parallel algorithm for locally clustering trajectories.

Nonetheless, to the best of our knowledge, there does not exist a parallel GPU algorithm for clustering trajectories locally. The only work related to our proposed algorithm TraclusGPU is G-DBSCAN [ARMS+13], which is a parallel algorithm for DBSCAN clustering [EKSX96][TSK05]. This work differs from TraclusGPU in that G-DBSCAN is designed for density-based clustering of trajectories, while TraclusGPU performs

local clustering of trajectories (i.e., TraclusGPU does not cluster the trajectories as whole, but instead it partitions them into segments and clusters these segments). In other words, both algorithms cluster objects of a different nature.

5.2 *Overview of TraclusGPU*

TraclusGPU is a parallel GPU algorithm for performing local trajectory clustering, and is based on the ideas of the Traclus algorithm, which is a serial algorithm for local trajectory clustering. TraclusGPU receives as input parameters a set S of trajectories, a positive integer *minPts*, and a positive real number $\varepsilon > 0$, which are the same three parameters of the Traclus algorithm.

TraclusGPU, just like Traclus, consists of three stages: the partitioning stage, the trajectory clustering stage, and the representative trajectory search stage. In the partitioning stage, TraclusGPU uses the Minimum Description Length Principle (MDL) to sub-divide (partition) the trajectories into segments [TSK05]. In the trajectory clustering stage, TraclusGPU clusters the resulting segments using a segment distance [LLHW07]. Finally, in the representative trajectory stage, TraclusGPU constructs a trajectory representative for each cluster.

The key idea behind TraclusGPU arises from the observation that the most time consuming stage of the serial Traclus algorithm is the segment clustering stage. For this reason, the main contribution of TraclusGPU consists in adapting the GPU parallelization ideas for the DBSCAN algorithm [ARMS+13] (which is also based in the BFS algorithm presented in [HN07]) to segments by arranging the segment data in a

linear fashion, so as to ensure global memory coalescing. Also part of the modifications on the work [ARMS+13] consists in being able to classify the segments into border, core and noise points [EKSX96].

5.3 *Description of TraclusGPU*

In this section we explain the details of TraclusGPU. Its pseudocode is presented in Figure 31. TraclusGPU consists of a partitioning stage, a local trajectory clustering stage and a representative trajectory finding stage. As can be seen in Figure 31, TraclusGPU receives as inputs a set of trajectories *S*, a positive integer *minPts*, and a positive real number ϵ.

5.3.1 Partitioning Stage

To partition the trajectories, TraclusGPU follows the same theoretical ideas first presented in [LLHW07], which consist of partitioning the trajectories according to the Minimum Description Length Principle. This is done by calling the function *Approximate Trajectory Partitioning*, described in [LLHW07], in parallel on a multicore CPU (Lines 2 – 4 in Figure 31). The reason for running *Approximate Trajectory Partitioning* on a multicore CPU is that this function exhibits thread divergence on a GPU (see Section I.2.4 which describes this GPU phenomenon), which entails a significant performance penalty. There is, however, no performance penalty when running it on a multicore CPU.

After partitioning trajectories into segments, segments are stored in four arrays: *beginPointX, beginPointY, endPointX,* and *endPointY*. These arrays satisfy that the i-th segment starts at the point *(beginPointX[i], beginPointY[i])* and ends with the point *(endPointX[i], endPointY[i])*. In this manner, global memory access to the segments can be done in a coalesced manner.

Now, because we perform coarse-grained parallelism at the function call level, there are no modifications to the internal section of the *Approximate Trajectory Partitioning* function, which already exists in the literature. This trajectory approximate trajectory partitioning algorithm was first introduced in [LLHW07], and is based on the Minimum Description Length (MDL) optimization principle. This algorithm consists in splitting the input trajectory into segments, such that the resulting segments is as close as possible to the input trajectory (this is called preciseness [LLHW07]), and such that the total number of resulting segments is as small as possible (called conciseness [LLHW07]).

5.3.2 Local Trajectory Clustering Stage

TraclusGPU calls the function CLUSTERSEGMENTSGPU (Line 5 in Figure 31), which is in charge of locally clustering the segments that resulted from Stage 1 of the algorithm. The first goal in this function is to generate a graph $G = (V, E)$ such that V consists of all the segments obtained in the partitioning stage, and (v, w) belongs to E if and only if segments v and w are within an ϵ segment distance. To represent this graph in the GPU we use the Compressed Sparse Row (CSR) format [BFGM+09] because it is a concise

representation for sparse graphs (as opposed to an adjacency matrix), and because it arranges its elements in arrays, which helps memory coalescing on a GPU. This function assigns a GPU thread for every pair of segments *(i,j)* and computes their segment distance in parallel. If this distance is less than the ϵ value, the corresponding segments are considered neighbors (Lines 12—18). Then, the algorithm computes the degree of each node (segment) in the graph by counting the number of neighbors (Lines 19—21). This operation can be done efficiently on the GPU with a sum reduction. After this, by performing an exclusive parallel prefix sum of the degrees of all nodes (Line 22) we can obtain the offsets in the adjacency list (in CSR form) for every node in the graph. Then, the algorithm fills in the adjacency list part of the CSR format (Lines 23—29).

After constructing the CSR representation of the segment adjacency graph, the algorithm proceeds to do a series of BFS traversals on the graph, until every node has been visited (Line 30 and Line 32). This is done with the MODIFIEDBFSGPU function, which first allocates two arrays *class* and *isSource* each of length equal to the number of nodes in the graph (number of segments). The *class* array is initialized with the value 'Noise,' indicating that so far all points are classified as noise points. The *isSource* array is initialized with *false* values except the first entry. This indicates that the algorithm will start exploring the immediate neighbors of its node 0. The remainder of this function somewhat resembles a BFS or a DFS graph traversal. Then the algorithm starts assigning a GPU thread for every node *i* such that *isSource[i]* is true (called the *source nodes*) (Line 37), and checks all the nodes adjacent to those source nodes (Line 38).

Each of these adjacent nodes is marked as visited (Line 40) and marked as a source node for the next iteration of the outermost while loop. If these adjacent nodes have a degree higher than *minPts* (Line 41) they are classified as *core*, otherwise, if the source node was classified as *border* or *core*, then the adjacent node (being adjacent to a border or core node) is classified as *border (Lines 44–46)*. Finally, the algorithm takes the arrays *class* and *cluster* and builds a list of clusters *C by grouping together all the nodes that belong to the same BFS/DFS tree. This is because nodes belonging to the same BFS/DFS tree are all reachable from each other, and therefore belong to the same cluster.*

5.3.3 Representative Trajectory Finding Stage

Once TraclusGPU has locally clustered all the segments, it calls the function *Representative Trajectory Generation* in parallel (on a multicore CPU) for every cluster. This function was first introduced in [LLHW07], and since our parallelism is coarse-grained we do not introduce any changes. This algorithm works by sweeping a vertical line through all the segments in a cluster, and then averaging the intersection points between the segments encountered and the vertical line [LLHW07].

```
Algorithm  Parallel local clustering of trajectories
Input: A set of trajectories S, minPts ∈ Z+, ε ∈ R+
Output: A set O of clusters, each with a representative trajectory.
 1: function TraclusGPU(S, minPts, ε)
    // Stage 1: Partition every trajectory into segments
 2:     for p ∈ S do in parallel
 3:         segs ← segs ∪ Approximate_Trajectory_Partitioning(p)
 4:     end for
    // Stage 2: Cluster the segments
 5:     clusters ← ClusterSegmentsGPU(segs)
    // Stage 3: Find the representative trajectories for each cluster
 6:     for c ∈ clusters do in parallel
 7:         c.repr ← Representative Trajectory Generation(c)
 8:     end for
 9:     return clusters
10: end function
11: function ClusterSegmentsGPU(segs)
12:     for (i, j) ∈ {0, ..., |segs| − 1}^2 do in parallel
13:         if dist(segs[i], segs[j]) < ε then
14:             isNeighbor_i[j] ← true
15:         else
16:             isNeighbor_i[j] ← false
17:         end if
18:     end for
19:     for i ∈ {0, ..., |segs| − 1} do in parallel
20:         degree[i] ← sum(isNeighbor_i)
21:     end for
22:     offsets ← ExclPfxSum(degree)
23:     for i ∈ {0, ..., |segs| − 1} do in parallel
24:         for j ∈ {0, ..., |segs| − 1} do
25:             if isNeighbor_i[j] then
26:                 adjList[offsets[i] + j] ← j
27:             end if
28:         end for
29:     end for
30:     return ModifiedBfsGpu(newGraph(adjList, offsets), degree)
31: end function
32: function ModifiedBfsGpu(G = (adjList, offsets), degree)
33:     class ← [Noise, Noise, ..., Noise]
34:     isSource ← [true, false, false, ..., false]
35:     visited ← [false, false, false, ..., false]
36:     while exist unvisited nodes do
37:         for i ∈ {0, ..., |G.V| − 1} and isSource[i] do in parallel
38:             for j ∈ {nodes adjacent to i} do
39:                 if !visited[j] then
40:                     (visited[j], isSource[j]) ← (true, true)
41:                     if degree[j] ≥ minPts then
42:                         (class[j], cluster[j]) ← (Core, i)
43:                     else
44:                         if class[i] = Core or class[i] = Border then
45:                             (class[j], cluster[j]) ← (Border, i)
46:                         end if
47:                     end if
48:                 end if
49:             end for
50:         end for
51:     end while
    // Using these arrays, this builds a list of clusters
52:     return BuildClusters(class, cluster)
53: end function
54: return TraclusGPU(S, minPts, ε)
```

Figure 31. Pseudocode of the TraclusGPU algorithm

CHAPTER IV
PERFORMANCE ANALYSIS

In this chapter we present the analyses of the worst-case work and space complexities of the proposed techniques, and also present extensive experimental studies of their performance in comparison with state of the art techniques.

1 Theoretical Analysis

1.1 *Complexity Analysis for TKSimGPU*

We now discuss the worst-case work and space complexity of the TKSimGPU algorithm given in Figure 19 in Chapter III by studying the worst-case complexity of each one of its functions.

We now estimate the total amount of work performed by the function ESTIMATE_ ε. This function computes the Hausdorff distance between every trajectory p in P_sample and every trajectory q in Q_sample, and then for every p in P_sample it sorts the associated list L_p. Since the worst-case work complexity of computing the Hausdorff distance between any two trajectories p and q is $O(|p| \cdot |q|)$, then computing the Hausdorff distance for every (p, q) in $P_sample \times Q_sample$ has worst-case work complexity $O(|P_sample| \cdot |Q_sample| \cdot \hat{p} \cdot \hat{q}) = O(|P_sample| \cdot |Q_sample|)$, where $\hat{p}$ and $\hat{q}$ are upper bounds to the sizes of the trajectories in P_sample and Q_sample, respectively. Then, the total amount of work done to sort one of the lists L_p is $O(|Q_sample| \cdot \log(|Q_sample|))$, so to sort all lists L_p for every p in P_sample the work complexity is $O(|P_sample| \cdot |Q_sample| \cdot \log(|Q_sample|))$. Therefore, the worst-case work complexity of Lines 8 to 12 of the algorithm in Figure 19 is

$O(|P_sample\ |\cdot|Q_sample|\cdot\log(|Q_sample|) + |P_sample|\cdot|Q_sample|) =$ $O(|P_sample\ |\cdot|Q_sample|\cdot log\ (Q_sample))$. Then in Line 13 the algorithm finds the average Hausdorff distance between each *p* in *P_sample* and the closest *K* trajectories in *Q_sample*, which has worst-case work complexity $O(|P_sample|\cdot K)$. Therefore, the total amount of work performed by the function ESTIMATE_ ε has worst-case time complexity $O(|P_sample|\cdot|Q_sample|\cdot\log(|Q_sample|) + |P_sample|\cdot K) = O(P_sample\cdot(|Q_sample|\cdot\log(|Q_sample|) + K))$. The worst-case space complexity of the function ESTIMATE_ ε is $O(P_sample\cdot Q_sample)$.

Next, we examine the total amount of work performed by the function NEAR-JOIN FILTER in Line 18 of Figure 19. To find the eMBR of a trajectory *p* the worst-case work complexity is $O(|p|)$, since we need to visit all points of *p* to determine its MBR. Then, to compute the eMBRs of all trajectories in *P* and in *Q*, we perform an amount of work of $O(|P|\cdot|Q|\cdot\hat{p}\cdot\hat{q})$. Here, since we are studying the worst-case work complexity, we assume that trajectories visit each grid cell (this assumption could hold in real-life, depending on the spatio-temporal distribution of the trajectories and how coarse the grid is). Therefore, Lines 19 and 20 of Figure 19 together have a combined worst-case work complexity of $O(|G|)$, where $|G|$ is the number of grid cells. Line 21 of Figure 19 has a worst-case work complexity of $O(|G|^2\cdot|P|\cdot|Q|)$. Then, to remove duplicates, the worst-case work complexity is $O(|G|^2\cdot|P|\cdot|Q|)$ if the array of candidate pairs is sorted in lexicographical order. Therefore, the total worst-case work complexity of the function NEAR-JOIN FILTER is $O(|P|\cdot|Q|\cdot\hat{p}\cdot\hat{q} + |G| + |G|^2\cdot|P|\cdot|Q|) = O(|P|\cdot|Q|\cdot\hat{p}\cdot\hat{q} + |G|^2\cdot|P|\cdot|Q|) = O(|P|\cdot|Q|\cdot(\hat{p}\cdot\hat{q} + |G|^2))$. The worst-case

space complexity of the function NEAR-JOIN FILTER is $O(P \cdot |G| + Q \cdot |G| + P \cdot Q) = O(P \cdot Q)$, if we assume that $|G|$ is constant.

Next, we examine the total amount of work performed by the function REFINE_TKSIMGPU. It sorts the candidates for every p in P, which has a worst case work complexity of $O(|P| \cdot |Q| \cdot \log(|Q|)$ if we assume that every trajectory in P retrieves all trajectories in Q as candidates. Then, REFINE_TKSIMGPU takes the most similar K trajectories for every p in P, and this has a worst-case work complexity of $O(|P| \cdot K)$. Then, the total complexity of this function is $O(|P| \cdot |Q| \cdot \log(|Q|) + |P| \cdot K) = O(|P| \cdot |Q| \cdot \log(|Q|))$ if we assume that $K < |Q|$. The worst-case space complexity of the function REFINE_TKSIMGPU is $O(|P| \cdot |Q|)$.

Finally, we turn our attention to the total amount of work performed by the function TOP-K TRAJECTORY SIMILARITY in Line 1 of Figure 19 in Chapter III. This function initially calls FILTER_TKSIMGPU, which computes a *P_sample* and a *Q_sample*, then at each iteration it calls the ESTIMATE_ ε and NEAR-JOIN FILTER functions. As we have analyzed earlier, this is an amount of work of order $O(P_sample \cdot (|Q_sample| \cdot \log(|Q_sample|) + K) + |P| \cdot |Q| \cdot (\hat{p} \cdot \hat{q} + |G|^2)$. Since P_sample and Q_sample are much smaller than $|P|$ and $|Q|$, we have that $O(P_sample \cdot (|Q_sample| \cdot \log(|Q_sample|) + K) + |P| \cdot |Q| \cdot (\hat{p} \cdot \hat{q} + |G|^2) = O(K + |P| \cdot |Q| \cdot (\hat{p} \cdot \hat{q} + |G|^2))$. Then, each iteration first counts the number of candidates for every p in *P_sample*, an operation which has worst-case work complexity $O(|P| \cdot |Q|)$ if all possible candidates are retrieved, and then it finds all those trajectories in P without K

candidates, which has worst-case work complexity $O(|P|)$. Finally, the function TOP-K TRAJECTORY SIMILARITY calls REFINE_TKSIMGPU. Hence, the total work complexity of this function is $O(I(K + |P| \cdot |Q| \cdot (\hat{p} \cdot \hat{q} + |G|^2)) + |P| \cdot |Q| + |P| + |P| \cdot |Q| \cdot \log(|Q|)) = O(I \cdot |P| \cdot |Q| \cdot (\hat{p} \cdot \hat{q} + |G|^2) + |P| \cdot |Q| \cdot \log(|Q|))$, where I is the number of iterations. If we assume that $|G|$ is constant, as are $\hat{p}$ and $\hat{q}$, then the worst-case work complexity of TKSimGPU is $O(|P| \cdot |Q| \cdot \log(|Q|))$, which is the same worst-case work complexity that we would obtain if we generate all pairs in $P \times Q$, then compute the Hausdorff distance for each pair, then sort for each p in P in increasing order of Hausdorff distance, and then take the first K candidates for every p in P, as in the naïveGPU algorithm. Overall, the worst-case space complexity of the TKSimGPU algorithm is $O(P \cdot Q)$, which is the same as naïveGPU's worst-case space complexity.

1.2 *Complexity Analysis for Top-KaBT*

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

We first estimate the total amount of work performed by Top-KaBT in the function HAUSDORFF_BOUNDS in Line 15 in Figure 22 in Chapter III. Because this function computes the lower and upper bounds of the Hausdorff distance between p and q for every (p,q) candidate pair in C, and since we know that, according to Observation III.3.12, the calculation of these lower and upper bounds has worst-case constant time complexity; therefore, the total amount of work done by the HAUSDORFF_BOUNDS function has worst-case time 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 $O(|C|)$.

Inside the function REMOVE_CANDIDATES, we see that in Lines 37 to 43 of Figure 22 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 algorithms 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 these lines simply require writing a 1 or a 0 for every candidate pair in C.

We now turn our attention to the function SORT_PRUNING. Let $\widehat{C_p}$ be an upper bound to the size of C_p for every p in P. We see that SORT_PRUNING requires sorting C_p for every p in the query set P. Therefore, this requires $O(|P| \cdot |\widehat{C_p}| \cdot log(\widehat{C_p}))$ amount of work to sort all the candidate sets C_p. SORT_PRUNING then eventually calls the functions HAUSDORFF_BOUNDS, FIND_CUT_POINT, and REMOVE_CANDIDATES, whose total amounts of work have already been calculated. We conclude then that the overall worst-

case work complexity of Top-KaBT is $O(|C| + |P| \cdot |\widehat{C_p}| \cdot log(\widehat{C_p}))$, and the worst-case space complexity is also $O(|C|)$.

1.3 *Complexity Analysis for TrajEstU*

We now discuss the worst-case work and space complexity of the TrajEstU algorithm. We shall first estimate the total amount of work performed by the function CONSTACCMODEL in Line 16 of Figure 30. This function first sorts the set *points,* with a worst-case work complexity of $(|points| \cdot \log(|points|))$. Then, it fits a constant acceleration model to a set *points*. Lines 21 to 27 of Figure 30 fill-in the entries of a matrix of size $(2 \cdot |points|)\times 6$ and a vector of size $(2 \cdot |points|)\times 1$, which corresponds to a total amount of work of $O(|points|)$. Then, in Line 28, the algorithm performs linear regression, which corresponds to a worst-case work complexity of $O(|points|)$. Therefore, the worst-case work complexity of the function CONSTACCMODEL is

$$O(|points| \cdot \log(|points|) + |points|) = O(|points| \cdot \log(|points)).$$

The worst-case space complexity of the function CONSTACCMODEL is $O(|points|)$ because both the matrix H and the vector Z use $O(|points|)$ space.

We compute now the total amount of work performed by the function ESTIMATETRAJECTORY in Line 30 of Figure 30. This function obtains an estimated true path out of a model. Its worst-case work complexity is $O(numPoints)$, where *numPoints* is the size of the trajectory estimation that we wish to obtain. This assumes that Line 35

takes *O(1)*. The function ESTIMATETRAJECTORY uses *O(|numPoints|)* space, where *numPoints* is the desired number of points of the estimation.

We now proceed to compute the total amount of work performed by the function FILLDATA in Line 1 of Figure 30. For every constant acceleration interval in Line 4, the worst-case complexity is *O(|clusterTree|)* for the range search in an R-tree [Gutt84][BKSS90] (however, the average-case complexity for a range search is *O(log(|clusterTree))* [BKSS90]), and then $O(|clusters_I| \cdot L)$, where L is an upper bound to the size of all trajectories. Therefore, the worst-case work complexity of the online portion of the FILLDATA algorithm is $O(|clusters_I| \cdot L \cdot |intervals|)$, where *|intervals|* is an upper bound to the number of constant acceleration intervals that trajectories have.

We know that the worst-case space complexity for computing the MBR of any trajectory is *O(L)*. The worst-case complexity for inserting a trajectory MBR into an R-tree is *O(NumTrajs)*, where *NumTrajs* is the number of elements in the tree. Therefore, before calling FILLDATA, we perform the worst-case amount of work which is $O(NumTrajs \cdot L)$.

1.4 *Complexity Analysis for TraclusGPU*

The function CLUSTERSEGMENTSGPU first computes the distances between all pairs of segments (Lines 12—18 of Figure 31), which has worst-case work complexity $O(|segs|^2)$. Then, this function performs a set of additions (Lines 19—21) of

complexity $O(|segs|)$, and a reduction (Line 22). Finally, this function outputs the adjacency list, which has a work complexity of $O(|segs|^2)$. Therefore, the function CLUSTERSEGMENTSGPU has an overall worst-case complexity of $O(|segs|^2)$. Also, since this function has worst-case space complexity $O(|segs|^2)$ (if the graph is dense).

The function MODIFIEDBFSGPU is similar to the work proposed in [HN07] but, unlike that work, our function MODIFIEDBFSGPU traverses the whole graph (while [HN07] only traverses the subgraph reachable from a single node) and also classifies points as core, border and noise. This function has a worst-case work complexity of $O(|V| \cdot |L| + |E|)$, where V is the set of vertices of the graph, E is its set of edges, and L is the number of levels (the number of iterations of the outer-most while loop) [LWH10]. In the worst case, $L = O(|V|)$, so that the worst-case work complexity of this function is $O(|V|^2) = O(|segs|^2)$. The worst-case space complexity of this function is $O(|segs|)$ because it stores three arrays of length $|segs|$ to store the output, and to keep track of the state of the algorithm.

The overall worst-case work complexity of TraclusGPU is then $O(|segs|^2 + |S| \cdot \lambda + |C| \cdot \rho)$, where S is the input trajectory set to TraclusGPU, λ is an upper bound to the size of all trajectories in S, C is the resulting number of clusters, and ρ is the cost of finding a representative trajectory for a cluster. The overall worst-case space complexity of TraclusGPU is then $O(|segs|^2)$.

2 Experimental Analysis

Note that in the following section we use *scientific e notation*, the same used in scientific calculators and programming languages, to represent large quantities. Examples of scientific e notation are 3e6, to denote 3×10^6, and 1.1e3, to denote 1.1×10^3.

2.1 *Experimental Analysis of TKSimGPU*

In this section we describe the experiments performed on our proposed TKSimGPU algorithm for processing top-K trajectory similarity queries on GPUs.

2.1.1 Experimental Setup

2.1.1.1 Hardware and Software Description

Our multicore CPU algorithm was implemented in C using OpenMP. Our GPU algorithm was implemented in C, using CUDA 6.5, Thrust 1.8 [HB10] and CUB 1.4.1 [Mer11], and our experiments were performed in a Ubuntu 14.04 workstation equipped with two six-core Intel Xeon E5 2620v2 chips running at 2.1GHz, 64GB of DDR3 RAM and an Nvidia Quadro K5000 GPU with 4GB of RAM.

2.1.1.2 Datasets and experiment setup

For our experiments we use the GeoLife dataset [ZXM10] of real trajectories. The GeoLife dataset contains 17,621 trajectories whose lengths add up to 1,251,654 kilometers, and span an interval of 48,203 hours. The total number of points *(x,y,t)* in the trajectories of the GeoLife data set is 23,667,828. These trajectories were collected with the use of GPS phones and GPS loggers by Microsoft Research Asia.

For our experiments we have selected the subset of all these trajectories that are labeled with the keyword "walk." The reason for this is that these trajectories are shorter and hence there is less dead space (the empty area inside an MBR) within the MBRs of the trajectories. From these trajectories we have also removed all those trajectories that consist of only a single point (and have MBR with area 0) because those are not interesting trajectories. We have segmented each one of the trajectories of the original trajectory set by splitting a trajectory if the object describing the trajectory is stationary for more than 30 minutes (similar to what is done in the literature [RDTD+15]). We also split the original trajectories to ensure that no resulting trajectory has more than 256 points. The reason for this is that, since we are using MBRs for filtering, a very long trajectory could potentially span the whole space and its corresponding MBR would be the size of the whole space and would not help during the filtering stage. We end up with a total of 18,000,000 tuples *(x,y,t)* belonging to 86,648 trajectories, which we keep in the GPU's global memory.

2.1.1.3 Competing Algorithms

In these experiments we compare an implementation of TKSimGPU on a GPU and an implementation of a naïve exhaustive search on GPU, which we call *naïveGPU*. The naïve exhaustive search algorithm for processing top-K trajectory similarity queries finds the Hausdorff distances between all pairs of $(p, q) \in P \times Q$, and then sorts those distances to select for every $p \in P$ the top K most similar trajectories in Q using the function REFINE_TKSIMGPU (see Line 43 in Figure 19).

Both the GPU implementation of TKSimGPU and the GPU implementation of the naïve exhaustive search algorithm run with 512 threads per thread block.

2.1.1.4 Experimental Parameters

We now describe the types of parameters of the following set of experiments. These parameters are divided into two classes: static parameters and dynamic parameters. The static parameters are not changed in all experiments. The dynamic parameters, on the other hand, may have their values changed in an experiment. The way this is done is follows. In each experiment one dynamic parameter is chosen as the study parameter, and then we study the impact of that parameter on the performance of the algorithms. This study parameter will then assume different values in a given interval, while all the other dynamic parameters are kept constant at their default values. We will now describe the parameters of our experiments, which are summarized in Table 3.

One of the dynamic parameters is the size of the query trajectory set ($|P|$), which assumes values in the range from 20 to 100 trajectories, and whose default value is 60, which is the mean of that interval. Then there is the size of the database ($|Q|$), assuming values in the range from 28,000 to 56,000 trajectories, and that has a default value of 40,000, which is close to the mean of that interval. The last dynamic parameter is the value of K, which lies in the interval from 10 to 160, with a default value of 70.

The static parameters of this set of experiments are the size of the grid and the size of the sample used by TKSimGPU. In both cases, we used the values that yielded the best performance.

Parameter Name	**Geolife Data**	
	Range of Values	**Default Value**
Size of the query trajectory set |P|	20 – 100 Trajectories	60 Trajectories
Size of the database trajectory |Q|	28,000 – 56,000 Trajectories	40,000 Trajectories
K	10 – 160	70
Grid Size	128×128	128×128
Sample Size	512	512

Table 3. Experimental parameters of TKSimGPU and Top-KaBT

2.1.1.5 Performance Metrics

The performance metric used is *average query execution time* (ET). We measure the time it takes our algorithms to process a query from the instant when it is issued, until the instant when the query finishes executing.

2.1.2 Experimental Results

2.1.2.1 Impact of the query set size

In this experiment we use a database (*Q*) of size 40,000 trajectories (containing 10,330,000 data points), and $K = 70$. Figure 32 shows the experiment results. The labels in the horizontal axis are given in the format x(y), where x is the number of trajectories in the query set (*P*) and y is the total number of points contained in *P*. For example, the label “100 Tr (17e3 Pt)” indicates that the size of *P* is 100 trajectories, and if we sum up all the points contained in those trajectories we have 17,000 points. We observe that TKSimGPU is 3.37x faster than the GPU naïve exhaustive search algorithm. The reason

for this is that TKSimGPU's filtering stage is able to produce a candidate result set that has a size equal to 29.7% the size of the set $P \times Q$ on average. On the other hand, the naïve GPU algorithm has to exhaustively find all the Hausdorff distances for all pairs in $P \times Q$. From Figure 2 we can also observe that the query execution time of our GPU TKSimGPU implementation is approximately linear in the size of the query set (*P*), and we can also verify the fact that the naïve GPU implementation must be linear in the size of the query set (*P*) if the database size is fixed.

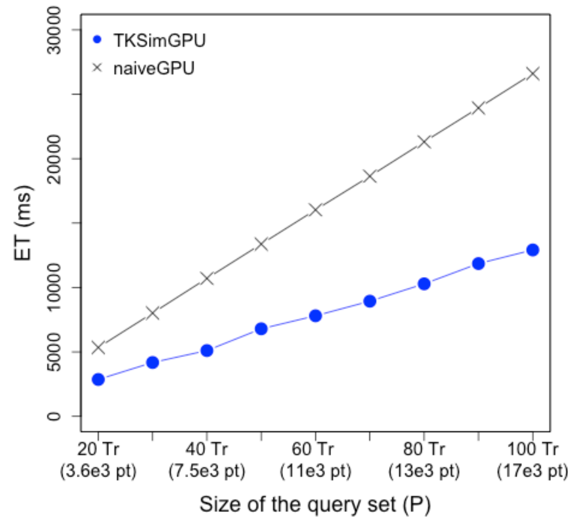

Figure 32. Query set size vs. execution time (TKSimGPU)

2.1.2.2 *Impact of the database size*

In this experiment we use a query set (*P*) of size 60 trajectories and choose *K = 70*. Figure 34 shows the results of this experiment. The labels in the horizontal axis are given in the format x(y), where x is the number of trajectories in the database (*Q*) and y is the total number of points contained in *Q*. For example, the label "36e3 Tr (7.7e6 Pt)" indicates that the size of *Q* is 36,000 trajectories, and if we sum up all the points contained in those trajectories we have 7,700,000 points. In this figure we observe that

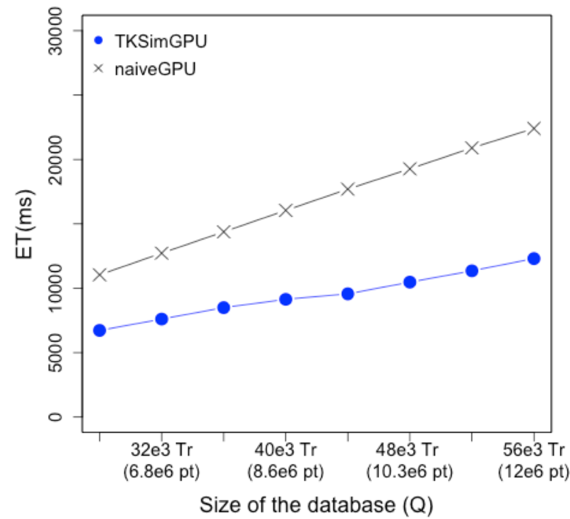

Figure 34. Database size vs. execution time (TKSimGPU)

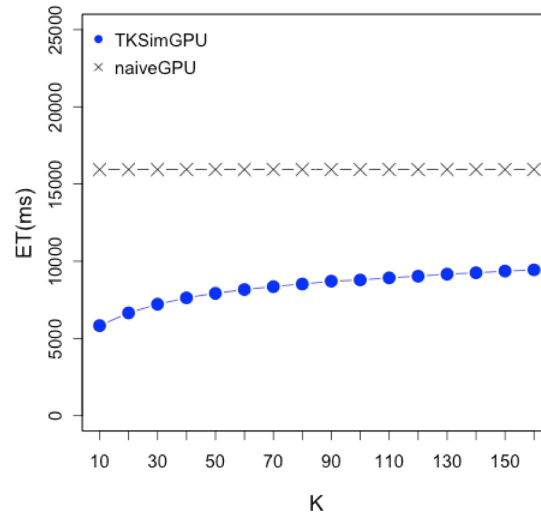

Figure 33. K vs. execution time (TKSimGPU)

our GPU implementation is 1.72x faster than the naïve GPU algorithm. The reason why this speedup is smaller than the one in the previous experiment (where TKSimGPU is 3.37x faster than the naïve GPU) is because in this experiment the size of the sample is kept fixed at 512, but at the same time the database set is increasing with 4,000 elements

at a time. The consequence of this is that, as the size of the database grows, the sample trajectory set drawn is less representative of the set *Q*, and this leads to a poorer (larger) epsilon, which in turn leads to candidate sets that are larger than they need to be. Again, we confirm that the naïve GPU algorithm has a linear complexity on the size of the database because the query set is kept fixed.

2.1.2.3 Impact of K

In this experiment we use a query set (*P*) of size 60 trajectories, a database of size 40,000 trajectories (10,330,000 data points) and vary *K* from 10 to 160. Figure 33 shows the results of this experiment. In this figure we observe that TKSimGPU is 1.42x faster than the naïve GPU algorithm on average. The experiment also shows that the query execution time of the naïve GPU algorithm remains practically constant if we increase the value of *K*. The reason for this behavior is that the most time-consuming tasks of the naïve implementation, which are writing $P \times Q$ to global memory, then finding the Hausdorff distance between every pair in $P \times Q$, and then sorting this set, do not depend on *K* at all. The only stage of naïveGPU that does depend on *K* is the function REFINE_TKSIMGPU (see Line 43 in Figure 19), which is basically a parallel copy between two global memory arrays and is very inexpensive, as the above results show. On the other hand, in Figure 33 we observe that the query execution time of our GPU implementation of TKSimGPU does depend on *K* because the choosing of the epsilon value for performing the near-join filter uses *K* as a parameter. A larger value of *K* will lead to a larger epsilon value, and that is the reason why TKSimGPU exhibits this behavior. However, we also observe that, as we increase *K*, the query processing time increases rather slowly, and this is because the size of the database (*Q*) is kept fixed, so

having a fixed size for the trajectory database samples under a constant size of Q will not degrade how representative the samples are of Q's spatial distribution, and hence, how good our epsilons are.

2.1.2.4 Conclusions of TKSimGPU's Experimental Results

Our conclusions from the experimental evaluation of TKSimGPU are the following:

- Existing parallel GPU trajectory similarity query processing algorithms like Gowanlock and Casanova's [GC14][GC16] and U2STRA[ZYG12] are not applicable for processing top-K trajectory similarity queries. Therefore, there does not exist a parallel GPU algorithm for processing top-K trajectory similarity queries.
- TKSimGPU is the first parallel GPU algorithm for processing top-K trajectory similarity queries.
- TKSimGPU performed significantly better (3.37x faster execution time) than the existing naïveGPU implementation on GPUs using a real-world large-scale dataset.
- Our experiments on a real-world large-scale dataset showed that the size of the trajectory query set is linear in the overall query execution time.
- The size of the database has an almost linear impact in the overall query execution time when running TKSimGPU on a real-world large-scale dataset.
- Our experiments on the Geolife dataset show that K has a sub-linear impact on the execution time. This is evidenced in the fact that the rate of increase of the

query execution time decreases as K grows larger. This means that TKSimGPU scales well with K.

2.2 *Experimental Analysis of Top-KaBT*

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 GPU, TKSimGPU [LGZY15], when combined with Top-KaBT to reduce candidate sets against TKSimGPU itself and against a naïve exhaustive GPU search algorithm. The naïve exhaustive search algorithm finds the Hausdorff distances between all pairs of $(p,q) \subseteq P \times Q$, and then sorts those distances to select the top K most similar trajectories in Q for every $p \in P$.

2.2.1 Experimental Setup

2.2.1.1 Hardware and Software Description

For our experimental evaluation of Top-KaBT, we use the same hardware and software environment described in Section 2.1.1.2.

2.2.1.2 Datasets and experiment setup

For our experimental evaluation of Top-KaBT, we use the same Geolife dataset described in Section 2.1.1.2.

2.2.1.3 Competing Algorithms

In these experiments we compare our proposed candidate trajectory pair pruning technique, Top-KaBT, combined with TKSimGPU against TKSimGPU alone (with no pruning help from Top-KaBT) and against a naïve exhaustive search GPU algorithm called *naïveGPU*. NaïveGPU is a parallel GPU 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.

2.2.1.4 Performance Metrics

For these experiments we use the following performance metrics: average query execution time (measured in milliseconds) and the percentage of candidate trajectory 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 $P \times Q$.

2.2.1.5 Experimental Parameters

For this experiment we have used the exact same parameters of the experimental evaluation of TKSimGPU, presented in Section 2.1.1.4. We avoid repeating them here in this section.

2.2.2 Experimental Results

2.2.2.1 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)).

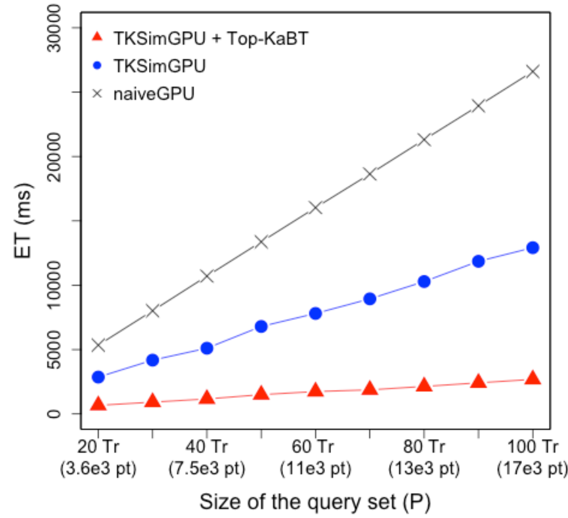

Figure 35. Query set size vs. execution time (Top-KaBT)

In Figure 35 we see that the average query execution times of all three techniques seems 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 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 function of the size of P.

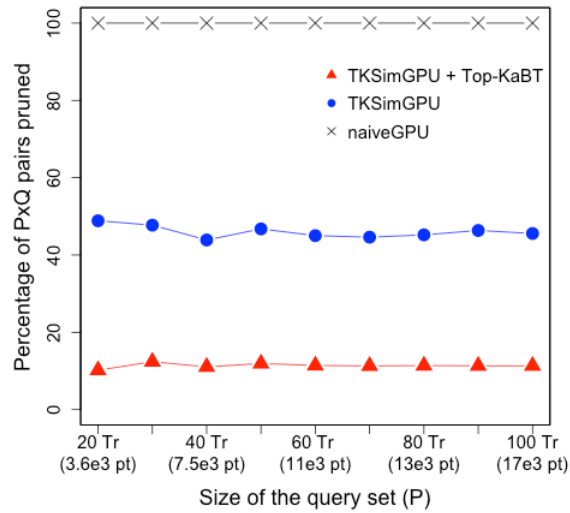

Figure 36. Query set size vs. % candidate pairs explored (Top-KaBT)

In Figure 35 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 36, 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 naïveGPU's.

Figure 36 shows the impact of the size of the query set ($|P|$) on the percentage of pairs $P \times Q$ 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 $P \times Q$, as expected. In Figure 36 we also observe that for all three algorithms the percentage of *(p,q)* candidate pairs in $P \times Q$ 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 $P \times Q$, but the number of candidate pairs pruned also increases by n (because Theorem III.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 36.

The previous observation is also consistent with Figure 35, 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 37 we observe the impact of the size of the query set ($|P|$) on the average

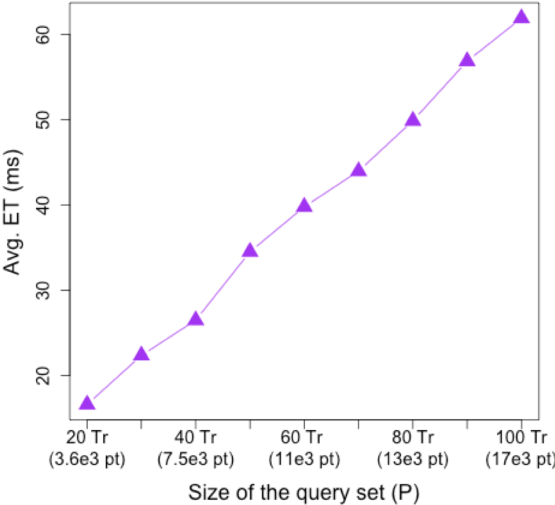

Figure 37. Query set size vs. execution time of Top-KaBT alone

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 35 and Figure 37 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.

2.2.2.2 *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 5,000,000 points up to 12,000,000 points (x,y,t)).

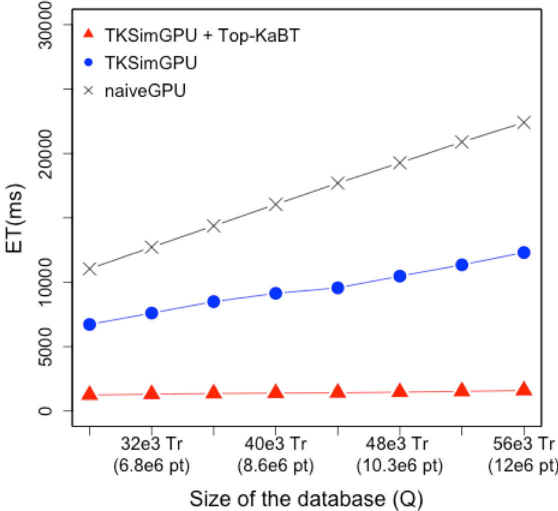

Figure 38. Database size vs. query execution time

In Figure 38 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 38 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 $P \times Q$, while TKSimGPU performs pruning and thus reduces the size of the candidate pairs set.

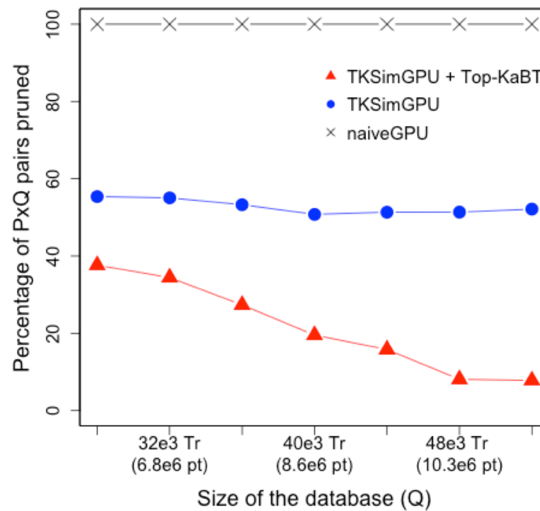

Figure 39. Database size vs. % candidate pairs explored

Figure 39 shows the impact of the database size ($|Q|$) on the percentage of candidate pairs in $P \times Q$ 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 q_0, q_1, q_2 with lower bounds (for their respective Hausdorff distances to p) 1, 2 and 3, respectively, and with upper bounds (for their respective Hausdorff distances to p) 3, 2, and 4, respectively. Then, according to the definition of a cut point, 1 is a cut point associated with the candidate set of p. Now, consider another trajectory q_4 in the database with lower bound 2.5 and upper bound 4. If q_4 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. However, if a trajectory q_5 with lower bound (for its distance to p) 0.5 and upper bound 0.75 is added to the set of candidate pairs of p instead of q_4, 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 would impact the percentage of candidate pairs pruned depends on the spatial distribution of the dataset.

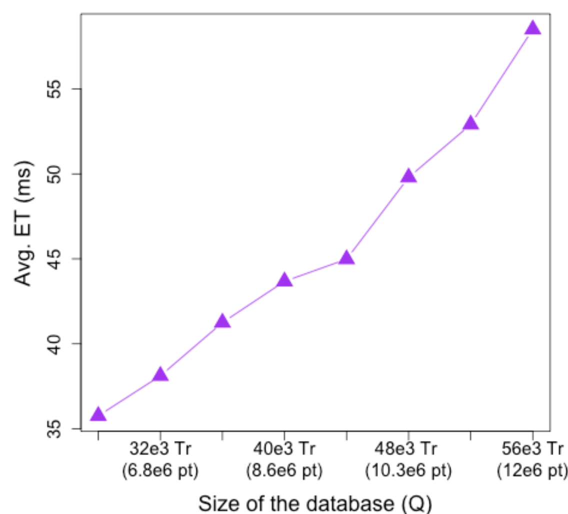

Figure 40. Database size vs. execution time of Top-KaBT alone

In Figure 40 we observe a similar behavior to the one in Figure 37, where the average execution time of 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.

2.2.2.3 *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 41 we observe that the average query execution time of the naïveGPU 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

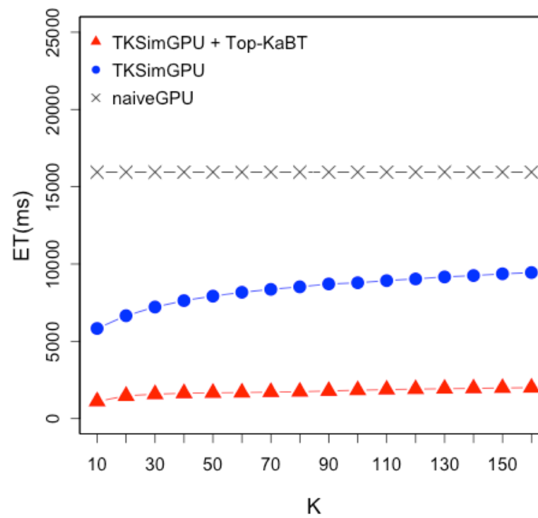

Figure 41. K vs. query execution time

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 41 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.

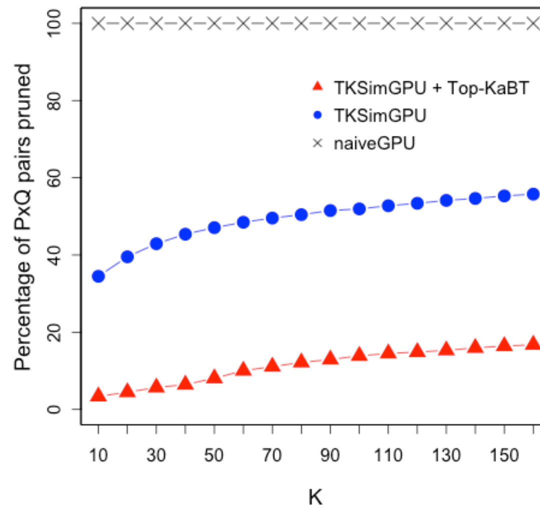

Figure 42. K vs. % candidate pairs explored

In Figure 42 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 turn is 4 times smaller than that of naïveGPU.

In Figure 43 we observe that the average execution time of Top-KaBT exhibits an overall tendency to increase as *K* grows. However, its behavior looks less like a straight line than in the case of Figure 37 and Figure 40. The reason for this is that when changing the value of *K* and leaving the sizes of the query set and the database 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 22 in Chapter III), the sorting of Q_p (Line 5 of Figure 22), the left shifting of the array (Line 6 of Figure 22), and the finding of the cut-points (Line 7 of Figure 22), 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 22 when the spurious candidate pairs are removed.

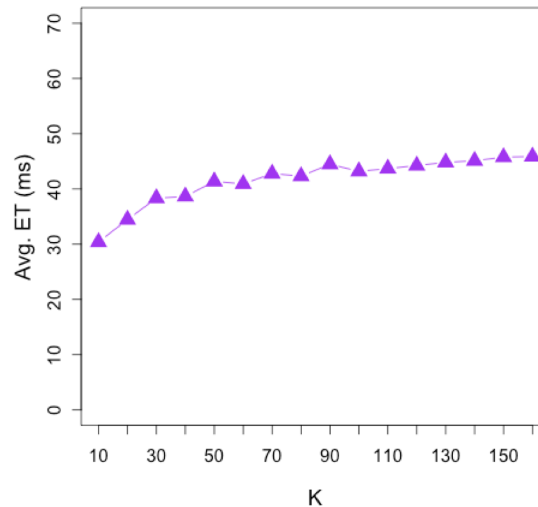

Figure 43. K vs. execution time of Top-KaBT alone

Since larger values of K usually lead to a lower percentage of candidate pairs pruned, larger values of K require more write operations in Line 49, which leads to slightly longer execution times, as can be seen in Figure 43.

2.2.2.4 Conclusions of Top-KaBT's Experimental Results

Our conclusions from the experimental evaluation of Top-KaBT are the following:

- Top-KaBT is a pruning technique to help reduce the size of the candidate sets generated by top-K trajectory similarity query processing algorithms, and is applicable for any such query processing algorithm that uses a similarity measure based on the triangular inequality.
- The execution time of Top-KaBT alone, i.e., the execution time of running only the pruning algorithm and not running the entire top-K trajectory query processing algorithm, is negligible for a real-world large-scale dataset like Geolife.
- When dealing with any real-world large-scale dataset, such as Geolife, the pruning performance of Top-KaBT (a measure of how much work Top-KaBT saves) shows very little impact as the size of the trajectory query set increases. Hence, Top-KaBT can be used in applications where it is desirable to retrieve similarities for a large number of query trajectories.
- The pruning performance of Top-KaBT in the Geolife dataset scales well in terms of the database size. However, this pruning performance in terms of the database size is, unlike the case of the performance for the trajectory query set size, depends on the spatial distribution of the dataset.

- Similarly, the pruning performance of Top-KaBT in the Geolife dataset scales well in terms of *K*. This means that Top-KaBT can be used in applications where the users want to retrieve large amounts of similar trajectories for every query trajectory.
- The execution time of Top-KaBT scales well as the size of the trajectory query set increases. For this reason, Top-KaBT can be used in applications where users desire to obtain similar trajectories for a large number of query trajectories.
- The execution time of Top-KaBT scales well as the value of *K* increases. Therefore, Top-KaBT can be used for a wide variety of applications that demand large values of *K*.
- Top-KaBT does not require any user-defined parameters. Hence, a Top-KaBT user does not need to search in a large parameter space for the parameter values that yield the best performance for Top-KaBT. For this reason, Top-KaBT can be easily applied as an auxiliary pruning tool for any top-K trajectory similarity query processing algorithm that uses a metric satisfying the triangular inequality.

2.3 *Experimental Analysis of TrajEstU*

2.3.1 Experimental Setup

2.3.1.1 Hardware and Software Description

The algorithms used in these experiments were implemented in Java 8, and were run on a workstation equipped with an Intel Xeon E5 and 64GB of RAM.

2.3.1.2 Datasets and Experiment Setup

For our experiments, we use two real datasets and one synthetic dataset. The first real dataset used is the deer dataset collected by the Starkey project [LHW07] consisting of the trajectories of deer from 1993 to 1996, and obtained through radio-telemetry. This dataset has 32 trajectories and 20,000+ points. The second real dataset is the hurricane dataset [LHW07] consisting of the trajectories of Atlantic Hurricanes occurring during the period from 1950 to 2004. This dataset contains 608 trajectories and 19,000+ points. Since the real life datasets are small, we generated a larger synthetic dataset to test the scalability of our technique. This synthetic dataset consists of 100,000 trajectories whose points sum up to 10,000,000, and was generated using moveHMM [MLP16], which simulates animal trajectories. These trajectories correspond to movements in unconstrained spaces.

To obtain the ground truth data, we assume that all the trajectories in the database are the ground truth data, i.e., the trajectories in the database are considered correct with no uncertainty.

Now we discuss how we generate the input trajectories whose true paths we wish to estimate. Input trajectories are trajectories with uncertainty for which we want to compute estimates for its positions. To generate our input trajectories, we randomly remove high sampling rate trajectories from the database and then add Gaussian noise with distribution $N(0, \sigma^2)$ to every one of its sample points, in order to simulate measurement uncertainty. Then, to simulate the model uncertainty, a subset of the

sample points is removed from these trajectories to ensure a low-sampling rate. We artificially add timestamps to all the points of all trajectories, and that timestamp added to each point is its index in its corresponding trajectory. Therefore, this scheme assumes that the trajectories in the datasets have a uniform sampling rate. For example, to generate an input trajectory with half the sampling rate of a given ground truth trajectory, we remove every other point from the latter. The resulting input trajectory is then said to have a *sampling rate* of 0.5 because its sampling rate is half of that of the ground truth.

2.3.1.3 Competing Algorithms

In these experiments we compare our proposed technique, TrajEstU, against Chazal et al.'s technique [CCGJ+11], described in Section III.2.2.2. This technique, like TrajEstU, is a data-driven technique that exploits the underlying database in order to reduce the model and measurement uncertainty in a trajectory. It does so by embedding the noisy input trajectory and the non-noisy database trajectories into a higher-dimensional space. Once this is done, the technique moves each point of the input trajectory towards the nearest embedded database points, and then brings all the resulting points back into the native space of the trajectory.

2.3.1.4 Experimental Parameters

The dynamic and static parameters of the following experiments are given in Table 4. One of the dynamic parameters is the sampling rate. We measure the sampling rate as a real number in the interval [0,1] for all datasets, where a value of 0.5, for example,

expresses that the input trajectory was obtained from the ground truth trajectory by removing every other point; hence, the resulting input trajectory has half the number of points of the ground truth trajectory. A value of 0.3 for the sampling rate of the input trajectory expresses that this trajectory was generated from the ground truth trajectory by keeping one point, then removing the next two points, then keeping the next one, then removing the next two, and so on. Therefore, a value of 1 for the sampling rate of the input trajectory expresses that the input trajectory has the same points as the ground truth. For this dynamic parameter we chose 0.5 as the default value for all datasets because it lies in the middle of the range of values. Another dynamic parameter in this set of experiments is the length of the input trajectory (the summation of the distances between consecutive points in a trajectory). The range of values of this parameter naturally depends on the dataset. However, in all cases we chose a default value equal to the average of the lengths of all trajectories in each dataset. So, for example, for the deer dataset, we chose a default value of 10,000 because that is the average of the lengths of all trajectories in that dataset. A third dynamic parameter for our experiments is the standard deviation of the measurement noise. For all datasets, we chose a range of values between 0m and 30m because we assume that the trajectory points are collected through GPS sensors, and these sensors have errors less than 20m in 99% of the cases, and errors less than 6m in 96% of the cases [GPS17]. This is also the reason why we chose 6m as the default value for this parameter. Our final dynamic parameter for our experiments is the acceleration tolerance, which is used by TrajEstU to split an input trajectory into near-constant acceleration intervals. For this parameter we have chosen the value that produced the highest accuracy for TrajEstU.

Besides the dynamic parameters, we have a static one, belonging to TrajEstU, whose value is kept constant in all experiments. This is the epsilon eMBR, which delimits the size of the area used by TrajEstU to search for clusters that are close to a given input trajectory. For this parameter, we have chosen the values that produced the highest accuracy for the technique. There are also two static parameters belonging to Chazal et al.'s algorithm, which are the number of nearest neighbors in the embedded space that are averaged (*K*), and the number of adjacent trajectory points that are combined into one tuple in order to embed points into a higher-dimensional space (*n*). For these parameters we have chosen the values that yielded the highest accuracy for this algorithm.

Parameter Name	**Deer Dataset**		**Hurricane Dataset**		**Synthetic Dataset**	
	Range of Values	**Default Value**	**Range of Values**	**Default Value**	**Range of Values**	**Default Value**
Sampling Rate	0.1 – 1.0	0.5	0.1 – 1.0	0.5	0.1 – 1.0	0.5
Input Trajectory Length	3,000 – 15,000	10,000	100-600	300	100 – 500	300
Standard deviation of the measurement noise	0 – 30	6	0 – 30	6	0 – 30	6
Epsilon eMBR	20	20	20	20	20	20
Acceleration tolerance	0.1 – 100	1.1	0.1 – 100	1.1	0.1 – 100	1.1
K	5	5	5	5	5	5
n	1	1	1	1	1	1

Table 4. Experimental parameters of TrajEstU

2.3.1.5 Performance Metrics

To measure the estimation *accuracy* we use the EDR trajectory similarity measure [COO05] to determine the similarity between the trajectory suggested by our algorithm and the ground truth. The EDR trajectory similarity between two trajectories q_1 and q_2 is similar to the edit distance on strings in that it computes the minimum number of points that have to be modified in order to transform one of the input trajectories into the other. The key difference between EDR and the edit distance lies in the matching function: in EDR two points match if they are within a distance of δ, where δ is a fixed positive real number. For calculating the EDR distance, we use $\delta = 20$m. We see then that the larger the distance, the more dissimilar q_1 and q_2 are. Therefore, the maximum possible distance is $max(|q_1|, |q_2|)$ where $|q_i|$ is the number of points in the trajectory q_i. Using this, we define the *EDR match percentage* between q_1 and q_2 as $(1 - EDR(q_1, q_2)) / max(|q_1|, |q_2|)$. Hence, the EDR match percentage is a number from 0 to 1, and the higher the EDR match percentage, the more similar q_1 and q_2 are.

Our second evaluation metric is the *average query execution time* (ET), where we measure the time from the moment when the query starts executing until it finishes. The average execution time is taken over 30 runs of the same query.

2.3.2 Experimental Results

2.3.2.1 Impact of the Sampling Rate

We study the effects of the model uncertainty by studying the impact of the sampling rate of the points in the input trajectory. This is because the lower the sampling rate, the higher the model uncertainty is. In these experiments pertaining to the impact of the

sampling rate, the x-axis denotes the sampling rate multiplier. To simulate different sampling rates, we choose a set of input trajectories, called the *original set of input trajectories*, and then remove points from them to obtain other input trajectories with lower sampling rates. The value 1 of the sampling rate multiplier refers to the case where we use the original set of input trajectories, the value 1/2 refers to the case where we use the input trajectory set resulting from removing every other point from the original input trajectories, and the value 1/3 refers to the case where we use the input trajectory set resulting from removing every other three points from the original input trajectories, and so on. Therefore, the higher the value of the sampling rate multiplier, the higher the sampling rate.

Figure 44 shows the impact of the sampling rate on the accuracy of both algorithms for the deer dataset. The figure shows that the lower the sample rate, the lower the accuracy

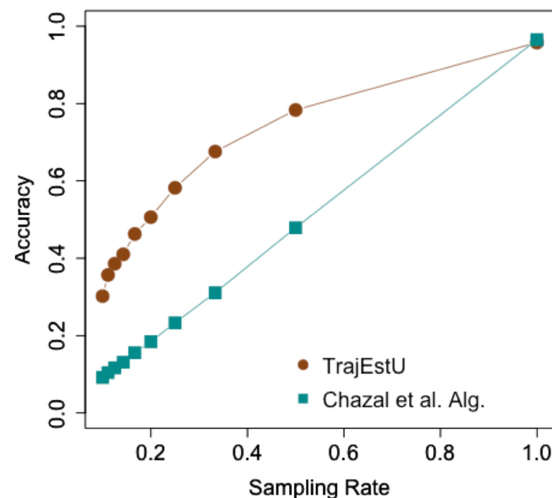

Figure 44. Sampling rate vs. accuracy (deer dataset)

is. This is because a lower sampling rate implies higher model uncertainty, which in turn leads to more difficulty in accurately predicting the true path of the moving object. We observe that TrajEstU's accuracy is consistently higher than Chazal et al.'s algorithm. We also observe that TrajEstU's accuracy seems to increase exponentially, which can be explained because the number of points of the query trajectories also increases exponentially.

Figure 45 shows the impact of the sampling rate on the execution time of both algorithms for the deer dataset. In this figure we observe that the execution time increases with a larger sampling rate because the bulk of the work done by the online phase of TrajEstU is proportional to the number of sampled points considered when building the linear regression models. For this dataset, the execution time of both algorithms exhibit a similar (sort of linear) behavior as the sampling rate changes, and also the average execution times for both algorithms are fairly close to each other, with TrajEstU running slightly faster (although the difference is negligible).

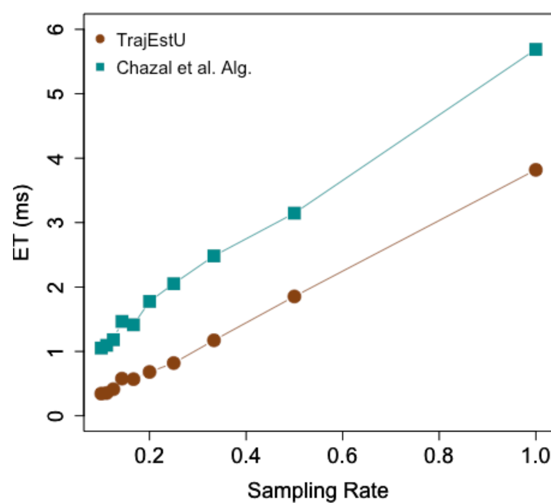

Figure 45. Sampling rate vs. execution time (deer dataset)

Figure 46 shows the impact of the sampling rate on TrajEstU's accuracy for the hurricane dataset. This figure also shows that the lower the sample rate, the lower the accuracy is, which is an expected behavior. In this figure we also observe that both algorithms have identical behaviors to those exhibited in the deer dataset: TrajEstU has a consistently higher accuracy, and as the sampling rate increases, the accuracy increases in a non-linear fashion because the number of points in the input trajectory varies also in a non-linear manner.

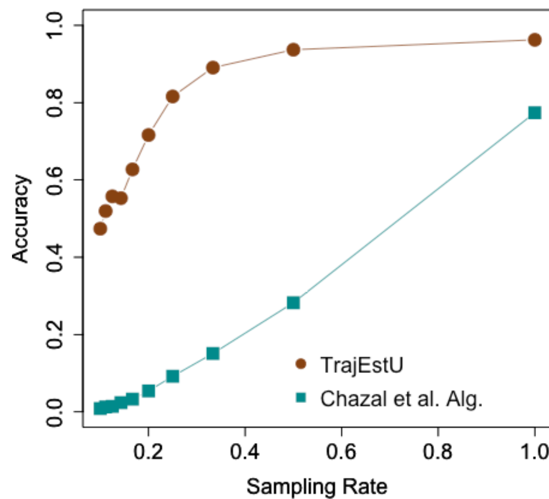

Figure 46. Sampling rate vs. accuracy (hurricane dataset)

Figure 47 shows the impact of the sampling rate on the average execution time of both competing techniques for the hurricane dataset. In this experiment we see that both algorithms exhibit comparable execution times (the differences in their ETs is in the order of few milliseconds, which is not substantial), with TrajEstU being only slightly faster.

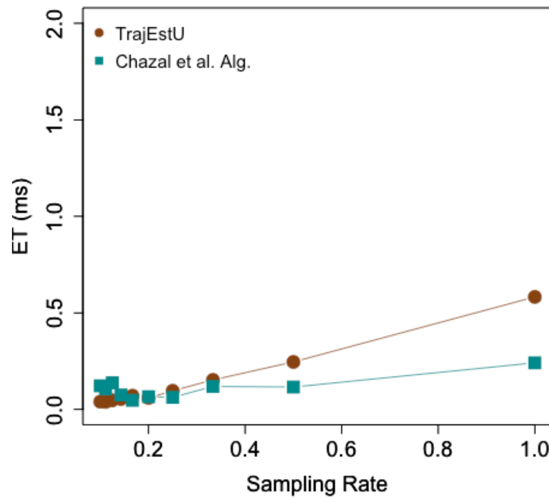

Figure 47. Sampling rate vs. execution time (hurricane dataset)

Moreover, in Figure 45 and Figure 47 we observe that for the deer and hurricane datasets the execution time of Chazal et al.'s algorithm is shorter than that of TrajEstU; however, in the synthetic dataset the opposite happens and TrajEstU is the faster executing algorithm. When studying the impact of the sampling rate on the execution time of both algorithms (Figure 45, Figure 47 and Figure 49**)**, we cannot conclude that either algorithm is consistently faster than the other. The reason for why sometimes TrajEstU is faster and sometimes slower than Chazal et al.'s algorithm is that the ETs of algorithms depend on the spatial distribution of the dataset. In the case of TrajEstU, this is because the execution time is proportional to the number of clusters found in a region; and in the case of Chazal et al.'s algorithm this is because the execution time depends on how expensive it is to find the k nearest neighbors for every embedded point of the input trajectory. Nonetheless, in all three datasets both algorithms are competitive in terms of execution time since the execution times are in the order of milliseconds. In other words, the differences in their execution times are negligible.

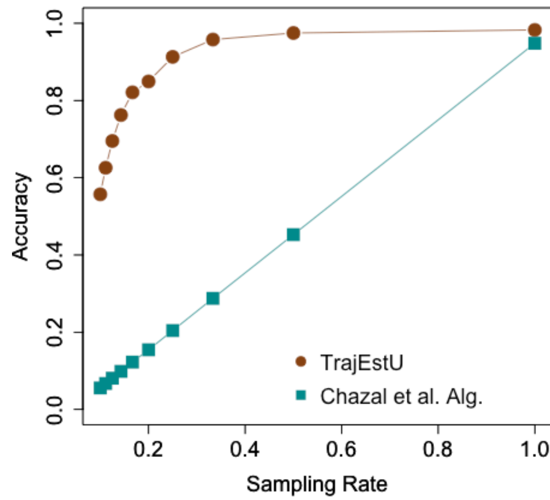

Figure 48. Sampling rate vs. accuracy (synthetic dataset)

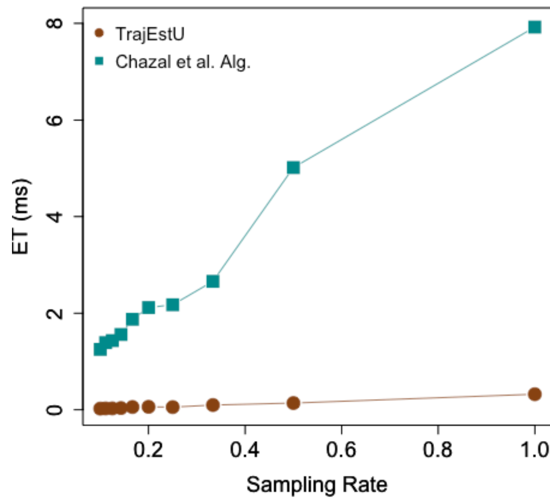

Figure 49. Sampling rate vs. execution time (synthetic dataset)

2.3.2.2 Impact of the Query Length

Figure 50**,** Figure 52 and Figure 54 show that the length of the input trajectory does not have a significant impact on the accuracy of either technique. This is expected because TrajEstU works by splitting the trajectory into intervals where the acceleration does not

change significantly, and the number and duration of these constant-acceleration intervals is not a function of the length of the input trajectory. The same is true for Chazal et al.'s algorithm, which does not take the length or the size of the trajectory as an input parameter.

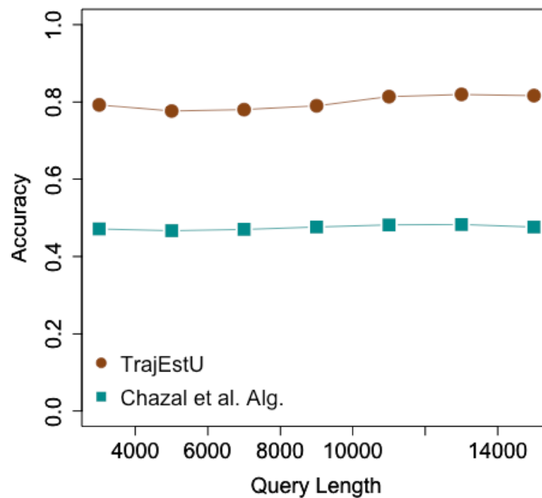

Figure 50. Query length vs. accuracy (deer dataset)

As we can see in Figure 51, Figure 53 and Figure 55, the execution time does, however, increase with the length of the input trajectory because, under a constant acceleration tolerance, the longer the trajectory, the greater the number of points and the greater the number of near-constant acceleration models that need to be fitted. The same is true for Chazal et al.'s algorithm because the work performed is proportional to the size of the trajectories, which is correlated with the length of the trajectory. In those figures we also observe that in some cases, like with the deer and synthetic datasets, Chazal et al.'s algorithm has shorter ET, while in other cases, like with the hurricane dataset, TrajEstU

has shorter ET. However, in all three datasets, we observe that both algorithms have extremely short execution times that are competitive with each other.

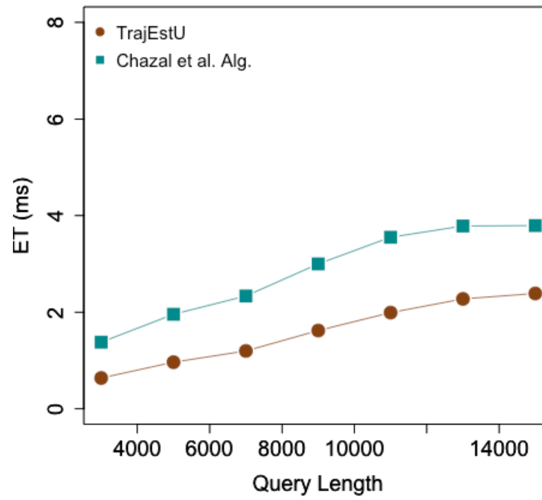

Figure 51. Query length vs. execution time (deer dataset)

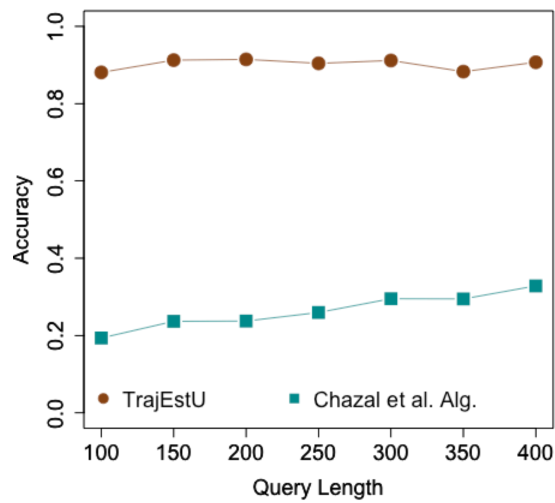

Figure 52. Query length vs. accuracy (hurricane dataset)

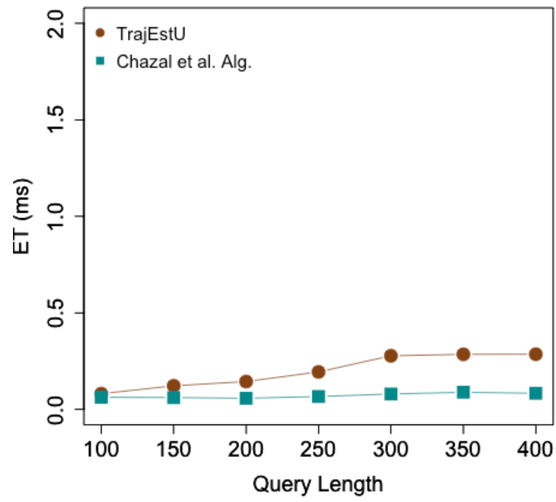

Figure 53. Query length vs. execution time (hurricane dataset)

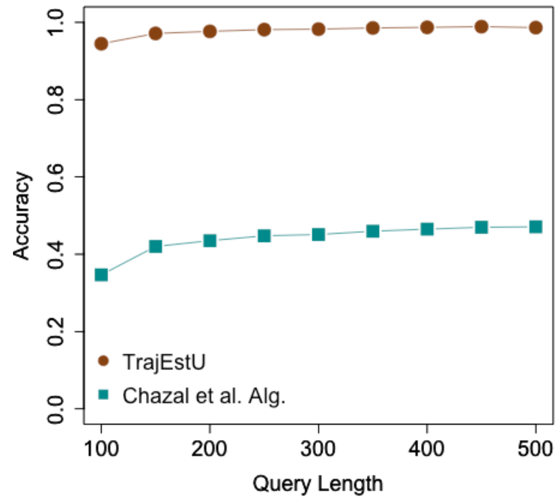

Figure 54. Query length vs. accuracy (synthetic dataset)

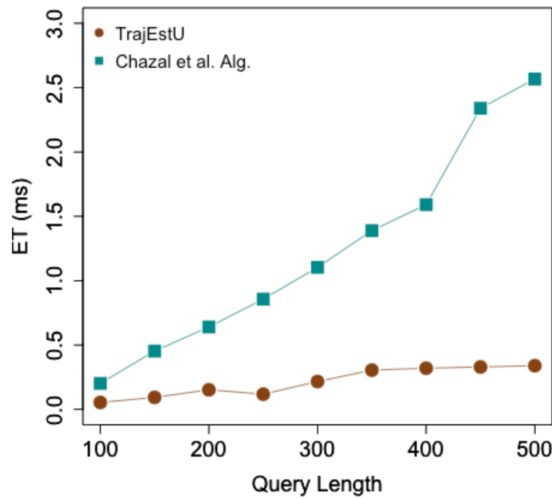

Figure 55. Query length vs. execution time (synthetic dataset)

2.3.2.3 Impact of the Standard Deviation of the Noise

In this experiment we study the impact of the magnitude of the variance of the Gaussian noise that was artificially added to every point in the query trajectories in order to simulate measurement uncertainty. Figure 56**,** Figure 58 and Figure 60 show that the greater the standard deviation of the measurement noise, the lower the accuracy is. This is expected because it becomes harder to accurately predict the true path of the moving object when the measurement/instrumentation uncertainty is higher.

In Figure 56 it can be seen that the accuracy of TrajEstU is up to 1.7X greater than that of the embedding algorithm, but the gap progressively narrows as the standard deviation of the noise increases, so that when the standard deviation is 30m, both algorithms exhibit the same accuracy. However, in practice, the standard deviation of the measurement error of GPS devices does not exceed 20m [GPS17], which means that in GPS applications TrajEstU would have better accuracy than the competing algorithm. A

similar behavior is also observed in the hurricane dataset, only that with this dataset TrajEstU exhibits up to 3.2X the accuracy of Chazal et al.'s algorithm, and, despite the fact that the gap also narrows as the noise increases, TrajEstU has consistently higher accuracy than the competing algorithm.

Figure 57, Figure 59, and Figure 61 show that the execution time is not impacted by the standard deviation of the measurement noise because when building the linear regression models, the amount of work performed by the technique is the same independently of this standard deviation of the measurement noise. Also, in Figure 57, Figure 59 and Figure 61 we observe that for all three datasets, the execution time of Chazal et al.'s algorithm shows no sensitivity towards the variation of the measurement noise. This is to be expected because the magnitude of the error does not play any role in the algorithm. We also observe that just like in all previous experiments, the execution times of both techniques are competitive with each other.

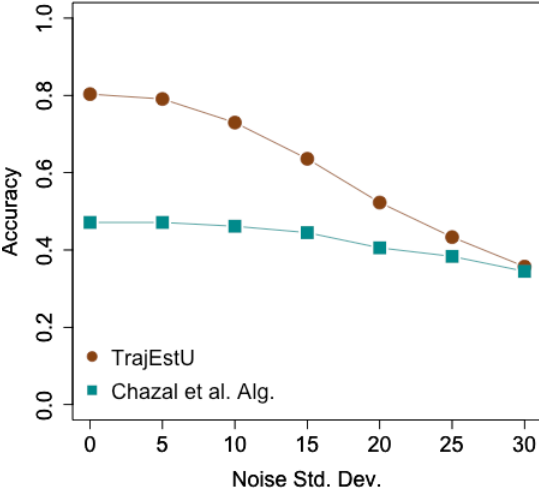

Figure 56. Standard deviation of the measurement noise vs. accuracy (deer dataset)

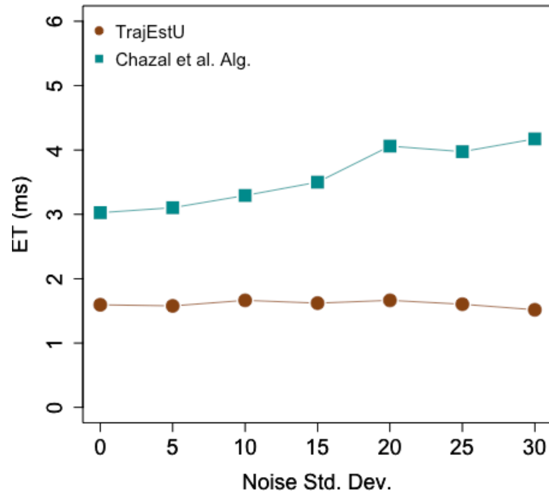

Figure 57. Standard deviation of the measurement noise vs. execution time (deer dataset)

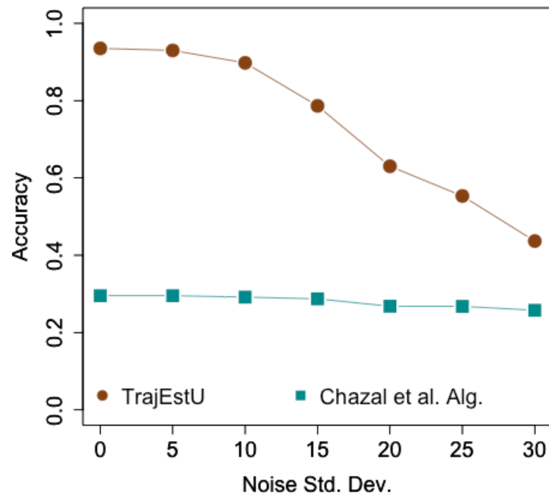

Figure 58. Standard deviation of the measurement noise vs. accuracy (hurricane dataset)

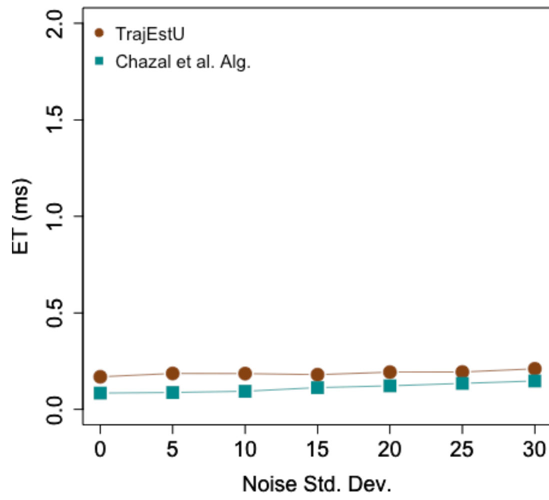

Figure 59. Standard deviation of the measurement noise vs. execution time (hurricane dataset)

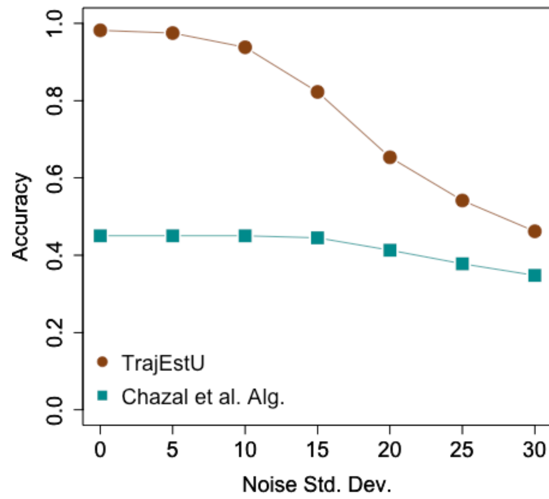

Figure 60. Standard deviation of the measurement noise vs. accuracy (synthetic dataset)

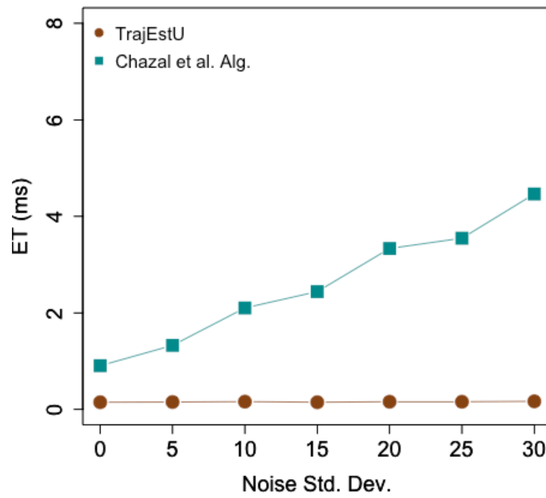

Figure 61. Standard deviation of the measurement noise vs. execution time (synthetic dataset)

2.3.2.4 Impact of the Acceleration Tolerance

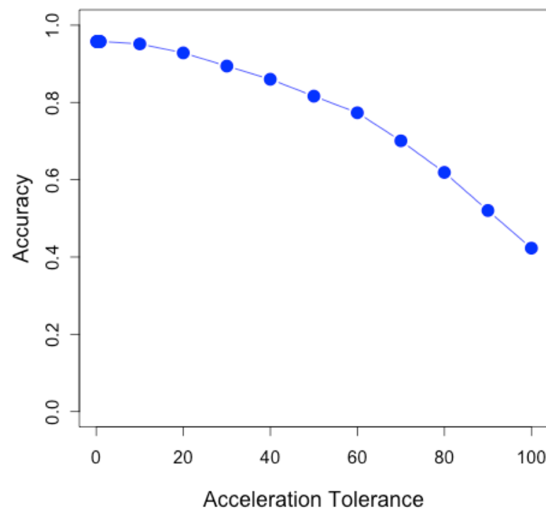

Figure 62.Acceleration tolerance vs. accuracy (deer dataset)

The acceleration tolerance is used to split a trajectory into near-constant acceleration intervals. Figure 62, Figure 63, and Figure 64 show that as the acceleration tolerance increases, the accuracy decreases. This is because as the acceleration tolerance is larger,

then within each near-constant acceleration interval, the acceleration varies more markedly so that our linear regression model, which assumes constant acceleration, cannot adequately fit the data.

2.3.2.5 Impact of the Dataset Size

Despite the fact that the synthetic dataset is significantly larger than the real life ones, we observe that the average query execution time for any given experiment did not change significantly between the datasets. This is because the dataset size only impacts the pre-processing stage where we cluster the trajectories, and this stage is performed off-line. The impact of the dataset size could then only influence the average query execution time through the number of resulting trajectory clusters that need to be considered in Line 7 of Figure 30 in Chapter III. In our three datasets, we observe that

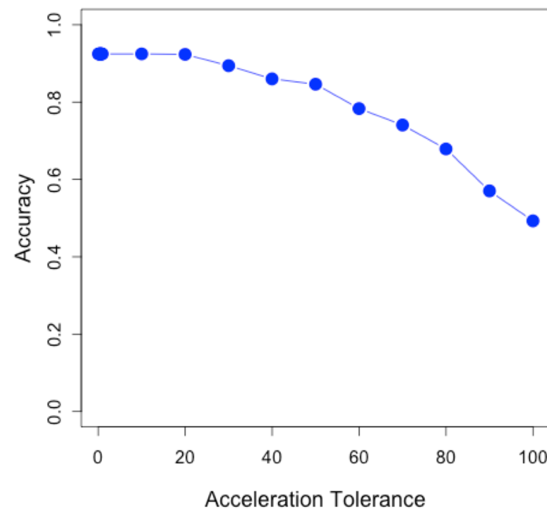

Figure 63. Acceleration tolerance vs. accuracy (hurricane dataset)

with the clustering parameters recommended by [LLLH10], the cluster density is the

same so that our algorithm considers a similar amount of clusters. From the figures in Section IV.2.3.2 that evaluate the impacts on the accuracy of TrajEstU we also observe that the accuracy did not vary much among datasets. This is expected in the deer and synthetic datasets because the latter dataset was generated using an animal movement simulator, and because the acceleration tolerance chosen led to constant acceleration intervals of about the same size; therefore, the linear regression models fitted sets of points with similar movement patterns and with about the same number of points.

2.3.2.6 Conclusions of TrajEstU's Experimental Results

Our conclusions from the experimental evaluation of TrajEstU are the following:

- TrajEstU is a pruning technique to help reduce the size of the candidate sets generated by top-K trajectory similarity.

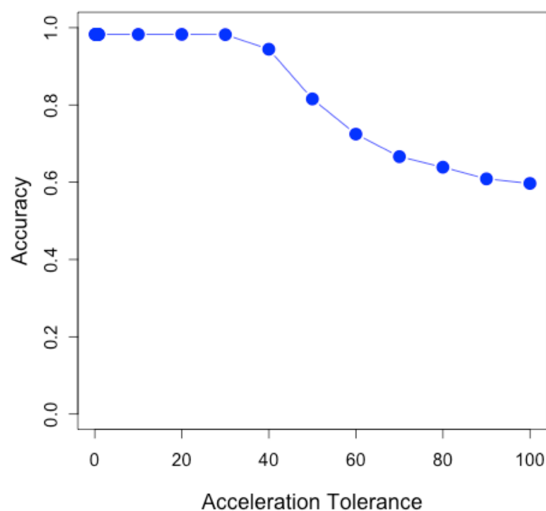

Figure 64. Acceleration tolerance vs. accuracy (synthetic dataset)

- TrajEstU achieves higher accuracy estimates than the state-of-the-art trajectory estimation techniques in unconstrained spaces.
- TrajEstU achieves a competitive execution time with existing trajectory estimation techniques on unconstrained spaces.
- TrajEstU's average execution time (for its online component) per input trajectory is negligible. Therefore, it can be used as a scalable pre-processing technique for trajectory estimation without any concerns that it will impose an execution time penalty.
- The most time-consuming portion of TrajEstU is the pre-processing stage, where trajectories are locally clustered according to Traclus [LHW07]. However, our experiments show that this algorithm can be efficiently parallelized with GPUs, which help diminish the pre-processing execution time.
- The accuracy advantage that TrajEstU holds against the competing technique narrows as the standard deviation of the measurement noise increases up to 20m. However, in real applications involving GPS sensors, the standard deviation of the noise is much smaller than 20m [GPS17], so this decrease in the accuracy advantage of TrajEstU is less significant in real-world applications.
- TrajEstU shows no accuracy impact as the size of the input trajectories increases. Therefore, TrajEstU can be used to estimate trajectories with large lengths without any concerns for decreases in the accuracy of the estimations.
- The execution time of TrajEstU scales well with the length of the input trajectories.

- TrajEstU shows no significant impact in accuracy in terms of the epsilon eMBR, nor in terms of the acceleration tolerance. Therefore, there should not be a big concern in finding the values for these parameters in order for TrajEstU to achieve its best performance.

2.4 *Experimental Analysis of TraclusGPU*

2.4.1 Experimental Setup

2.4.1.1 Hardware and Software Description

The algorithms described in this set of experiments were implemented in Java 8, on a workstation running Ubuntu 14.04, with two Intel Xeon E5 chips with six cores, 64GB of RAM, and an nVidia Quadro K5000 with 4GB of RAM.

2.4.1.2 Datasets and Experiment Setup

For our experiments, we use two a synthetic dataset consisting of 100,000 trajectories whose points sum up to 10,000,000, and that was generated using moveHMM [MLP16], which simulates animal trajectories.

2.4.1.3 Competing Algorithms

In this set of experiments we compare our proposed technique, TraclusGPU, against a serial implementation of the Traclus technique [LHW07].

2.4.1.4 Experimental Parameters

For our experiments we will see the impacts on the performance of TraclusGPU when we vary the values of the size of the set of trajectories that we wish to cluster, which we also refer to here as the size of the dataset. One reason for choosing this experimental parameter is that the main motivation for TraclusGPU was to solve the scalability problem posed by the serial Traclus technique, which took weeks to run on our dataset. Despite that TraclusGPU is a clustering algorithm, we do not study the quality of the clusters because TraclusGPU is a parallel version of Traclus, so the quality of the cluster it produces is the same as that of those produced by Traclus. Our static parameters in this experiment are ε and MinPts, which denote the size of the neighborhoods and the minimum number of segments in each cluster, respectively, in both Traclus and DBSCAN. Table 5 presents a summary of our experiment parameters.

Parameter Name	**Synthetic Data**	
	Range of Values	**Default Value**
Size of the trajectory set	10,000 – 70,000	35,000
$\boldsymbol{\varepsilon}$	2	2
MinPts	7	7

Table 5. Experimental parameters of TraclusGPU

2.4.1.5 Performance Metrics

Our evaluation metric is the *average query execution time* (ET), where we measure the time from the moment when the clustering algorithm starts executing until it finishes. The average execution time is taken over 30 runs of the same query.

2.4.2 Experimental Results

2.4.2.1 Impact of the Dataset Size

In this experiment we study the impact of the dataset size on the performance of both TraclusGPU and the serial Traclus algorithm. In Figure 65 we observe that there is a significant performance difference between TraclusGPU and the serial Traclus algorithm, where TraclusGPU is able to cluster a database with 4,000,000 segments (about 400,000 trajectories) in around 3 hours, while the serial Traclus does the same in sixteen hours. We see then that the performance increase in terms of execution time is around 5X.

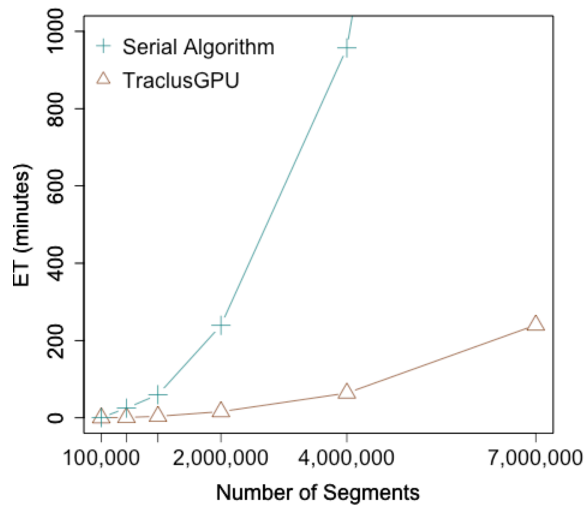

Figure 65. Number of segments vs. execution time (synthetic dataset)

2.4.2.2 Conclusions of TraclusGPU's experiment results

Our conclusions from the experimental analysis performed on TraclusGPU are the following:

- The total execution time of TraclusGPU significantly improves upon the execution time of the serial Traclus algorithm. In particular, TraclusGPU offered reasonable execution times of around 4 hours to cluster our large-scale dataset, as opposed to the serial Traclus algorithm, which took two weeks to run on the

same dataset. This shows the difference between a practical algorithm for Big Trajectory Data (TraclusGPU) and an impractical algorithm (Traclus).

- We observe that TraclusGPU has a comparable execution time to the serial Traclus algorithm when the total number of segments in the database is around 100,000 (around 10,000 trajectories) and that only when the number of segments grows beyond 2,000,0000 (around 200,000 trajectories) is that the execution time advantage of TraclusGPU really shows.

CHAPTER V
CONCLUSIONS AND FUTURE WORK

In this dissertation we have proposed a system for processing top-K trajectory similarity queries on Big Data using GPUs. This system consists of four components: TKSimGPU, the query processing engine, Top-KaBT, the parallel pruning technique that helps reduce the amount of work performed by the query processing engine, TrajEstU, the trajectory estimation technique to address the issue of uncertainty in trajectories, and TraclusGPU, the local trajectory clustering technique used to assist TrajEstU in addressing the uncertainty of trajectories.

The first component, TKSimGPU, is a parallel top-K trajectory similarity query processing technique on Big Data using GPUs. There are many applications for this type of query such as for social media trajectory sharing applications [ZXM10], where users are interested in finding potential friends with similar travel trajectories, ecology applications where scientists want to study the migration patterns of birds to help understand how diseases are transmitted between these animals [HGKL07], and in astronomy [GC14][GC16] where astronomers want to study the movements of galaxies.

The second component is Top-KaBT, which is our parallel pruning technique that is designed to help TKSimGPU better cope with the large volume of Big Trajectory data by removing spurious candidate trajectory pairs and their associated performance overhead.

The third component, TrajEstU, is our trajectory estimation technique, which was designed to help TKSimGPU deal with model and measurement uncertainty by building models for trajectories, out of which better trajectory estimates can be obtained.

The fourth component, TraclusGPU, is our local trajectory clustering technique, which is based on the ideas of the serial Traclus algorithm. TraclusGPU was designed to help address the scalability issue of the offline clustering technique, Traclus, used in TrajEstU when dealing with large datasets. This makes TrajEstU a suitable technique to cope with Big Trajectory data.

We conducted complexity analyses for all our proposed algorithms: TKSimGPU, Top-KaBT, TrajEstU and TraclusGPU. In addition to this, we performed experimental evaluations to compare TKSimGPU against an exhaustive search GPU algorithm, naïveGPU, in terms of query execution time. Then, we compared TKSimGPU against a combined approach TKSimGPU + Top-KaBT in terms of execution time, percentage of candidate trajectory pairs pruned. Finally, we compared our trajectory estimation technique TrajEstU against an existing state-of-the-art trajectory estimation algorithm, called Chazal et al.'s algorithm, in terms of accuracy and execution time.

1 Summary of the Performance Results

1.1 *Summary of the Results of TKSimGPU*

Processing top-K trajectory similarity queries poses many computational challenges. One of these challenges is the volume of the data. This is because in Big Trajectory

Data applications trajectories have many points, and the databases on which the search is performed have very large sizes. In addition to the large sizes of the trajectories and trajectory databases involved, there is the difficulty that computing the similarity between trajectories usually has a quadratic time complexity on the sizes of the trajectories, which makes processing top-K trajectory similarity queries an even harder problem.

In this dissertation we have proposed a parallel top-K trajectory similarity query processing algorithm for GPUs, called TKSimGPU. This algorithm is based on the idea that we can sample the trajectory query set and the trajectory database to help estimate the average trajectory similarity and then use this estimate to perform a series of near-join trajectory similarity queries until the query result set is complete. TKSimGPU was designed to deal with the volume characteristic of Big Trajectory Data through the use of parallelism. We now summarize the TKSimGPU as follows:

- To the best of our knowledge, TKSimGPU is the first parallel top-K trajectory similarity query processing algorithm on Big Data using GPUs.
- TKSimGPU is designed to deal with the volume characteristic of Big Trajectory data through the use of GPU parallelism. It assumes that trajectories have no uncertainty.
- TKSimGPU avoids an exhaustive search on the trajectory query set and the database by extracting random samples from both these sets, and then computing the average similarity between the samples. Then with this similarity

estimate, TKSimGPU performs a sequence of near-join trajectory similarity queries to obtain the final query result set.

- The linear data structures used by TKSimGPU were designed with the goal that memory accesses follow patterns that help ensure memory coalescing, thus guaranteeing good performance.
- TKSimGPU was designed with the goal of ensuring load balance across the thread blocks.
- The worst-case work complexity of TKSimGPU is $O(|P| \cdot |Q| \cdot \log(|Q|))$, where P and Q are the trajectory query set and the database set, respectively. This is the same complexity that most of the existing top-K trajectory similarity query techniques have (e.g., ERP [CN04], EDR [COO05]).
- The worst-case space complexity of TKSimGPU is $O(|P| \cdot |Q|)$.
- TKSimGPU performed significantly better than the existing naïveGPU implementation on GPUs using a real-world large-scale dataset.
- Our experiments on a real-world large-scale dataset showed that size of the trajectory query set is linear in the overall query execution time.
- Our experiments on a real-world large-scale dataset show that, despite the fact that the worst-case time complexity of TKSimGPU is $O(|P| \cdot |Q| \cdot \log(|Q|))|))$, where P and Q are the trajectory query set an the database set, respectively, the size of the database had an almost linear impact in the overall query execution time.
- Our experiments show that K seems to have a sub-linear impact on the execution time. This is evidenced in the fact that the rate of increase of the query execution

time decreases as K grows larger. This means that TKSimGPU scales well with K.

1.2 *Summary of the Results of Top-KaBT*

As we have mentioned, one of the issues of Big Trajectory data is the volume. This volume arises because of the large sizes of both the query trajectory sets and the trajectory database, and also because of the large number of points contained within the trajectories themselves. Our proposed top-K trajectory query processing algorithm, TKSimGPU, was designed to efficiently use GPUs in order to tackle the volume challenge. This algorithm, however, can still generate a large number of spurious candidate trajectory pairs that cannot form part of the result set, which leads to additional and unnecessary computational overhead. To help TKSimGPU better cope with the volume of Big Trajectory data, we proposed a GPU pruning technique, called Top-KaBT. Top-KaBT is 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 GPUs. Top-KaBT works by using only the lower and upper bounds of the similarity measure to remove candidate pairs that cannot belong to the query result set. This reduces the negative impact arising from the small size of the GPU'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 now summarize Top-KaBT as follows:

- To the best of our knowledge, Top-KaBT is the first GPU technique for pruning the candidate sets generated by top-K trajectory similarity query processing algorithms.
- Top-KaBT proved to be an effective and efficient pruning technique for removing spurious candidate trajectory pairs generated by top-K trajectory similarity query processing algorithms.
- One of the key ideas behind Top-KaBT is to compute a lower and an upper bound for the similarity measure for every candidate trajectory pair. It then uses these bounds to remove spurious candidate pairs that cannot form part of the query result set. By doing this, Top-KaBT performs a tradeoff: instead of computing the *expensive* similarity measure between all candidate trajectory pairs, it computes these *much cheaper* lower and upper bounds for all candidate pairs, and then computes the expensive similarity measure but on a reduced candidate trajectory set.
- One of the advantages of Top-KaBT is that it only makes the assumption that the underlying trajectory similarity query processing engine uses a similarity *metric*. Therefore, the ideas behind Top-KaBT can be applied not only for TKSimGPU, but also for other top-K trajectory similarity query processing algorithms.
- Another advantage of Top-KaBT is that it has no user-defined parameters. Therefore, when adding Top-KaBT as a pruning technique to a top-K trajectory similarity query processing engine, there is no need to search in a large

parameter space for the parameter values that yield the best performance results with Top-KaBT.

- The experiments performed on a real-world large-scale dataset showed that the execution time of Top-KaBT was negligible. Therefore, Top-KaBT has the advantage that it can be used with query processing engines without any concerns for execution time performance penalties.
- The worst-case work complexity of Top-KaBT is $O(|C| + |P| \cdot |\widehat{C_p}| \cdot log(\widehat{C_p}))$, where $|C|$ is the size of the candidate set generated by the query processing engine, $|P|$ is the size of the trajectory query set, and $\widehat{C_p}|$ is an upper bound to the size of the candidate trajectory sets of a single query trajectory *p*.
- The worst-case space complexity of Top-KaBT is $O(|C|)$, which is the size of the candidate set generated by the query processing engine.
- Our experiments show that both the pruning performance (i.e. the percentage of candidate pairs explored) and the execution time scale well in terms of the *K* parameter.
- Our experiments show that both the pruning performance (i.e. the percentage of candidate pairs explored) and the execution time scale well in terms of *P* the size of the query trajectory set.

1.3 *Summary of the Results of TrajEstU*

On top of the difficulty posed by the large volume of Big Trajectory data, trajectories can also be uncertain, which has a significant impact on the accuracy of the query results. In this dissertation we proposed a technique, called TrajEstU, for addressing the

issue of model and measurement uncertainty for trajectories moving in unconstrained outdoor space when processing top-K trajectory similarity queries on GPUs. TrajEstU works by splitting the lifetime of an object's trajectory into time intervals where the object's acceleration is nearly constant. Then TrajEstU uses the local trajectory clusters to obtain the movement patterns that are prevalent in the areas where trajectories have low-sampling rates, and linear regression to fit a constant acceleration model to the observed positions of the moving object. By using a linear regression model, TrajEstU reduces the uncertainty arising from the GPS measurements and the low-sampling rate of trajectories.

We now summarize TrajEstU as follows:

- TrajEstU is a technique for estimating the true paths of a trajectory considering model and measurement uncertainty.
- The experiments we performed on real and synthetic datasets show that the execution time of the online component of TrajEstU is negligible. Therefore, TrajEstU can be used without concerns for performance penalties.
- TrajEstU relies on an off-line pre-processing stage in which local trajectory clustering is done through the use of the Traclus algorithm [LHW07]. The reason why TrajEstU uses local trajectory clustering instead of regular trajectory clustering [Zheng15] is that we seek to estimate a trajectory around specific localities. If we perform regular trajectory clustering that would only result in the global or overall behavior of trajectories, which could be very different from that of the input trajectory that we wish to estimate.

- Our experiments also suggest that the offline component of TrajEstU can be efficiently implemented with the use of a hybrid multicore/GPU algorithm.
- TrajEstU performs better than the state-of-the-art trajectory estimation algorithms for both real and synthetic datasets, and for also small and large-scale datasets.
- TrajEstU not only possesses better accuracy than its competing algorithms, but it also has a comparable online execution time.
- The accuracy of TrajEstU decreases as the magnitude of standard deviation of the measurement noise increases.
- The accuracy of TrajEstU is an increasing function of the sampling rate, i.e., the higher the sampling rate, the higher the accuracy.
- Both the acceleration tolerance and the epsilon MBR parameters have little impact on the performance of TrajEstU. Therefore, there is not a big concern for searching for the parameter values that yield the best performance out of TrajEstU.
- The accuracy of TrajEstU is not sensitive to the length of the query trajectories. Therefore, TrajEstU can be used to accurately estimate Big Trajectory Data.
- The execution time of TrajEstU scales linearly with the length of the query trajectories, which helps confirm that TrajEstU can be used to efficiently and accurately estimate Big Trajectory Data.

1.4 *Summary of the results of TraclusGPU*

TrajEstU is our proposed technique to help our top-K trajectory similarity query processing technique deal with the issue of measurement and model uncertainty. The

online component of TrajEstU consists of building a set of linear near-constant regression models to help estimate the true path of an input trajectory. These linear regression models are built from the input trajectory points, as well as from the local behavior of trajectories that are located close to the input trajectory. To find this local behavior of trajectories, TrajEstU employs the serial Traclus algorithm [LLHW07] to locally cluster the trajectories in an off-line fashion. Nonetheless, our experiments on TrajEstU show that despite the fact that its on-line stages scale very well for Big Trajectory Data, its off-line stage (which is run just once for each trajectory database) that clusters the trajectory database using the existing serial trajectory clustering algorithm, Traclus, takes a considerable amount of time. For this reason, and in order to make TrajEstU practical for it to be applied to Big Trajectory Data applications, we proposed a parallel GPU algorithm to perform local trajectory clustering, called TraclusGPU, based on Traclus.

We now summarize TraclusGPU as follows:

- TraclusGPU is a parallel GPU algorithm for performing local trajectory clustering based on the ideas of Traclus serial technique.
- The total execution time of TraclusGPU significantly improves upon the execution time of the serial Traclus algorithm. In particular, TraclusGPU offered reasonable execution times of around 4 hours to cluster our large-scale dataset, as opposed to the serial Traclus algorithm, which took weeks to run on the same dataset. This shows the difference between a practical algorithm for Big Trajectory Data and an impractical algorithm.

- Our TraclusGPU algorithm scaled almost linearly with the number of processors used.

2 Future Research

In this section we discuss the future research directions related to the processing of trajectory similarity queries on Big Data using GPUs. We first discuss the future research directions related with each one of our proposed techniques: TKSimGPU, Top-KaBT, TrajEstU and TraclusGPU.

As our experiments suggest, TKSimGPU is an effective parallel algorithm for processing top-K trajectory similarity queries on GPUs, but it makes the assumption that both the trajectory query set and the database set fit in the GPU's global memory space. Nonetheless, with the large volume of Big Trajectory Data, this assumption does not hold. In the future, we would like to extend TKSimGPU to allow it to handle datasets that do not fit in the GPU's global memory space.

Another possible future research direction relates to TKSimGPU's similarity measure. TKSimGPU uses the Hausdorff distance as its trajectory similarity measure, which has many applications like urban planning [NJS11]. However, the Hausdorff distance does not take the temporal dimension into consideration when computing the similarity between trajectories. Taking the temporal dimension into account is useful for online travel trajectory sharing applications because 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. However, the Hausdorff distance

would not be able to distinguish trajectories that are very dissimilar in the temporal dimension. For this reason, we plan to design a parallel technique that uses a trajectory similarity measure that, unlike Hausdorff's, takes the temporal dimension into consideration.

Now, we comment on the future research directions concerning Top-KaBT. Although our experiments have shown that this is an efficient and scalable parallel pruning technique to reduce the size of the candidate sets of a top-K trajectory similarity query processing algorithm, it makes the assumption that its input candidate set resides in the device's global memory. Ideally, Top-KaBT would be integrated into the top-K trajectory query processing engine so that if a candidate trajectory pair is spurious, then it is never instantiated in memory. In this way, the spurious candidate pairs do not contend for the GPU's limited global memory space. In the future, we would like to extend Top-KaBT so as to avoid instantiating spurious candidate pairs in the GPU's global memory.

A second possible research avenue relates to the fact that Top-KaBT only makes the assumption that the underlying top-K trajectory similarity query processing engine uses any similarity metric. However, our experiments have not explored how Top-KaBT behaves using other trajectory similarity measures. For this reason, for our future work we would like to study how Top-KaBT behaves when using other trajectory similarity measures.

Although our experiments have shown that TraclusGPU significantly improves upon the serial Traclus algorithm in terms of execution time, there is still room for improvement. The reason for this is that the ET of TraclusGPU still grows rather quickly as the number of segments in the database increases, and because with 7,000,000 segments in the database, the ET of the algorithm is still non-negligible. Therefore, a future research direction related to TraclusGPU consists in improving the scalability of the algorithm to try to reduce its computational complexity.

So far, we have made the assumption that trajectories are constant and fixed at the beginning of query processing. However, trajectories are objects that grow in size (number of points) with time. Therefore, another possible future research direction is to extend our proposed system and techniques to deal with streaming trajectories, i.e., trajectories that are currently growing in size (number of points) as the query is being processed. An example of this query is "find the 2 birds that are currently flying with the most similar trajectories to a given bird in flight," and has applications when tracking objects in real time. Designing a technique to deal with these online queries is challenging for GPUs because to maximize the PCI-express bus throughput, one would ideally buffer the trajectory updates in the host's main memory before sending them through the PCI-express bus. This can lead to delays that could have an impact on the accuracy of the results.

Another research direction relates to designing parallel GPU techniques for trajectory outlier detection, i.e. finding trajectories in a database whose behavior markedly deviates from that of other trajectories in the database.

REFERENCES

[ARMS+13] Andrade, G., Ramos, G., Madeira, D., Sachetto, R., Ferreira, R., Rocha, L. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. In Proceedings of the International Conference on Computational Science (ICCS), 2013.

[Bayer71] Bayer, R. Binary B-Trees for Virtual Memory. In Proceedings of the ACM-SIGFIDET Workshop on Data Description, Access and Control, 1971.

[Bentley75] Bentley, J. L. Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9), 1975.

[BC94] Berndt, D., Clifford, J. Using dynamic time warping to find patterns in time series. In KDD workshop, pp. 359–370, 1994.

[Ble90] Blelloch, G.E. Prefix Sums and Their Applications. In John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, 1990.

[Bourgain85] Bourgain, J. On lipschitz embedding of finite metric spaces in hilbert space. Israel Journal of Mathematics, vol. 52, pp. 46–52, 1985.

[BCG02] Bruno, N., Chaudhuri, S., Gravano, L. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. ACM Transactions on Database Systems, 27(2), pp. 153–187, 2002.

[BDS14] Buchin, M., Dodge, S., Speckmann, B. Similarity of Trajectories taking into account geographic context. Journal of Spatial Information Science, 2014.

[BFJM+09] Buluç, A., Fineman, J. T., Frigo, M. Gilbert, J. R., Leiserson, C. E. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks (PDF). In Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 233—244, 2009.

[BKSS90] Beckmann, N., Kriegel, H. P., Schneider, R., Seeger, B. The R*-tree: an efficient and robust access method for points and rectangles. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 1990.

[CBPB10] Cagnacci, F., Boitani, L., Powell, R. A., Boyce, M. S. Animal ecology meets GPS-based radiotelemetry: a perfect storm of opportunities and challenges. Philosophical Transactions of the Royal Society B, 2010.

[CCGJ+11] Chazal, F., Chen, D., Guibas, L., Jiang, X., Sommer, C. Data-driven trajectory smoothing. In Proceedings of the ACM International Conference in Geographical Information Systems (ACM GIS), 2011.

[CLRS09] Cormen, T., Leiserson, C., Rivest, R., Stein, C. Introduction to Algorithms. The MIT Press, Third edition, 2009.

[COO05] Chen, L., Özsu, M. T., Oria, V. Robust and Fast Similarity Search for Moving Object Trajectories. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 491–502, 2005.

[CN04] Chen, L., Ng, R. On the Marriage of Lp-norms and Edit Distance. In Proceedings of the International Conference on Very Large Databases (VLDB '04), pp. 792–803, 2010.

[CPZ97] Ciaccia, P., Patella, M., Zezula, P. M-tree An Efficient Access Method for Similarity Search in Metric Spaces. In Proceedings of the International Conference on Very Large Databases (VLDB), 1997.

[CW06] Cao, H., Wolfson, O., Trajcevski, G. Spatio-temporal data reduction with deterministic error bounds. The VLDB Journal, vol. 15, no. 3, pp. 211–228, Sep. 2006.

[DTS08] Ding, H., Trajcevski, G., Scheuermann, P. Efficient similarity join of large sets of moving object trajectories. In *TIME '08*, pp. 79–87, 2008.

[DEKM98] Durbin, R., Eddy, S.R., Krogh, A., Mitchinson, G. Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, 1998.

[EM94] Eiter, T., Mannila, H. Computing discrete fréchet distance. In Technical Report CD-TR 94/64, Technische Universität Wien, 1994.

[EKSX96] Ester, M., Kriegel, H., Sander, J., Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD), pp. 226–231, 1996.

[FHL10] Fang, W., He, B., Luo, Q. Database Compression on Graphics Processors. In Proceedings of the VLDB Endowment, 3(1-2), pp. 670–680, 2010.

[FGT07] Frentzos, E., Gratsias, K., Theodoridis, Y. Index-based Most Similar Trajectory Search. In Proceedings of the International Conference on Data Engineering (ICDE), pp. 816–825, 2007.

[FRM94] Faloutsos, C., Ranganathan, M., Manolopoulos, Y. Fast subsequence matching in time-series databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 1994.

[GK10] Garland, M., Kirk, D. Understanding Throughput-oriented Architectures. Communications of the ACM, 53(11), pp. 58–66, 2010.

[Gor94] Gordon, N. Bayesian Methods for Tracking. Imperial College, University of London, London, 1994.

[GC14] Gowanlock, M., Casanova, H. Distance threshold similarity searches on spatiotemporal trajectories using GPGPUs. In Proceedings of High Performance Computing (HiPC). DOI: 10.1109 / HiPC.2014. 7116913, 2014.

[GC16] Gowanlock, M., Casanova, H. Distance threshold similarity searches: Efficient Trajectory Indexing on the GPU. In IEEE Transactions on Parallel and Distributed Systems (TPDS), 27(9), pp. 2533–2545, 2016.

[GM14] Gupta, S., Manchanda, R. TANH Spline Interpolation for Analytical Modelling of BK Ion Channels in Smooth Muscle. Asia Modelling Symposium, 2014.

[GPS17] National Coordination Office for Space-Based Positioning, Navigation, and Timing. GPS Accuracy. Retrieved on April 2, 2017 from the gps.gov website: http://www.gps.gov/systems/gps/performance/accuracy/

[Gutt84] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 1984.

[HB10] Hoberock, J., Bell, N. Thrust: A Parallel Template Library, http://thrust.github.io/, 2010.

[HGKL07] J. S. Horne, E. O. Garton, S. M. Krone, J. S. Lewis. Analyzing animal movements using brownian bridges. *Ecology*, vol. 88, no. 9, pp. 2354–2363, 2007.

[HP12] Hennessy, J. L., Patterson, D. A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2012.

[HSO07] Harris, M., Sengupta, S., Owens, J. D. Parallel prefix sum (scan) with CUDA. In H. Nguyen (ed.), GPU Gems 3, Addison-Wesley, 2007.

[HN07] Harish, P., Narayanan, P.J. Accelerating large graph algorithms on the GPU using CUDA. In IEEE High Performance Computing, pp. 197-208, 2007.

[Intel17] Product Specs of the Intel E7 8894 v4. Retrieved on April 2, 2017 from the Intel website: http://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz.

[JS07] Jacox, E.H., Samet, H. Spatial join techniques. ACM Transactions Database Systems, 32, 2007.

[K60] Kalman, R. E. A New Approach to Linear Filtering and Prediction Problems. Journal of Basic Engineering.,82:35, 1960.

[KP00] Keogh, E., Pazzani, M. Scaling up dynamic time warping for datamining applications. In Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD), pp. 285–289, 2000.

[KR05] Keogh, E., Ratanamahatana, C. Exact indexing of dynamic time warping. In Knowledge and Information Systems, 7(3), pp. 358–386, 2005.

[KH13] Kirk, D. B, Hwu, W. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kauffman, San Francisco, 2013.

[LGZY15] Leal, E., Gruenwald, L., Zhang, J., You, S. TkSimGPU: A Parallel Top-K Trajectory Similarity Query Processing Algorithm for GPGPUs. In Proceedings of the IEEE International Conference on Big Data (IEEE Big Data), pp. 461–469, 2015.

[LGZY16] Leal, E., Gruenwald, L., Zhang, J., You, S. Towards an Efficient Top-K Trajectory Similarity Query Processing Algorithm for Big Trajectory Data on GPGPUs. In Proceedings of the IEEE BigData Congress, 2016.

[LGZ16] Leal, E., Gruenwald, L., Zhang, J. Handling Uncertainty in Trajectories of Moving Objects in Unconstrained Outdoor Spaces. In Proceedings of the IEEE International Conference on Big Data (IEEE Big Data), 2016.

[LHW07] Lee, J., Han, J., Whang, K. Trajectory Clustering: A Partition-and- Group Framework. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2007.

[LKCD+10] Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. SIGARCH Comput. Archit. News, 38(3), pp. 451–460, 2010.

[LLD06] Lewis, M., Lakshmivarahan, S., Dhall, S.K. Dynamic Data Assimilation: A least squares approach. Cambridge University Press, 2006.

[LLLH10] Z. Li, J. Lee, X. Li, J. Han. Incremental Clustering for Trajectories. In Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA), 2010.

[LM13] Lustig, D., Martonosi, M. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 354–365, 2013.

[LWH10] Luo, L., Wong, M., Hwu, W. An Effective GPU Implementation of Breadth-First Search. In Proceedings of the Design Automation Conference, 2010.

[MLSC13] Ma, C., Lu, H., Shou, L., Chen, G. KSQ: Top-K Similarity Query on Uncertain Trajectories. In Proceedings of the IEEE Transactions on Knowledge and Data Engineering (TKDE), 25(9), pp. 2049–2062, 2013.

[Mer11] Merrill, D. Cuda Unbound (CUB), http://nvlabs.github.io/cub/, 2011.

[MLP16] Michelot, T., Langrock, R., Patterson, T.A. moveHMM: an R package for the statistical modelling of animal movement data using hidden markov models. Methods in Ecology and Evolution, 2016.

[NJS11] Nutanong, S., Jacox, E.H., Samet, H. An incremental Hausdorff distance calculation algorithm. In Proceedings of the VLDB Endowment, vol. 4., no. 8, pp. 506–517, 2010.

[Nvidia17] Product Specs of the Tesla K80. Retrieved on April 2, 2017 from the Nvidia website: http://www.nvidia.com/object/tesla-k80.html.

[P11] Pacheco, P.S. An Introduction to Parallel Programming. Morgan Kaufmann, 1st edition, 2011.

[PJT00] Pfoser, D., Jensen, C.S., Theodoridis, Y. Novel Approaches in Query Processing for Moving Object Trajectories. In Proceedings of the International Conference on Very Large Databases (VLDB), pp. 395–406, 2000.

[PKKF+11] Pelekis, N., Kopanakis, I., Kotsifakos, E. E., Frentzos, E., Theodoridis, Y. Clustering uncertain trajectories. Knowledge and Information Systems, 28(1), pp. 117—147, 2011.

[RDTD+15] Ranu, S., Deepak, P., Telang, A.D., Deshpande, P., Raghavan, S. Indexing and matching trajectories under inconsistent sampling rates. In Proceedings of the International Conference on Data Engineering (ICDE), pp. 999–1010, 2015.

[RRS00] Ramaswamy, S., Rastogi, R., Shim, K. Efficient Algorithms for Mining Outliers from Large Data Sets. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 427–438, 2000.

[SAMP+13] Sankararaman, S., Agarwal, P.K., Mølhave, T., Pan, J., Boedihardjo, A. P.

Model-driven matching and segmentation of trajectories. In *SIGSPATIAL*, pp. 234–243, 2013.

[TSK05] Tan, P., Steinbach, M., Kumar, V. Introduction to Data Mining. Pearson Prentice Hall, 1st edition, 2005.

[VGK02] Vlachos, M., Gunopoulos, D., Kollios, G. Discovering similar multidimensional trajectories. In Proceedings of the IEEE International Conference on Data Engineering (ICDE), 2002.

[VHK06] Vlachos, M. Hadjieleftheriou, M., Keogh, E. Indexing Multi-dimensional Time-Series with Support for Multiple Distance Measures. The VLDB Journal, 2006.

[VS98] Valdes-Perez, R., Stone, C. Systematic detection of subtle spatio-temporal patterns in time-lapse imaging: II. Particle migrations. Bioimaging, 6, 1998.

[WMDT+13] Wang, X., Mueen, A., Ding, H., Trajcevski, G., Scheuermann, P., Keogh, E. Experimental comparison of representation methods and distance measures for time series data. In Data Mining and Knowledge Discovery, 2013.

[WZX14] Y. Wang, Y. Zheng, Y. Xue. Travel time estimation of a path using sparse trajectories. In Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD), pp. 25–34, 2014.

[WZP12] Wei, L.Y., Zheng, Y., Peng, W.C. Constructing popular routes from uncertain trajectories. In Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD), pp. 195–203, 2012.

[W13] Wilt, N. The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley, 2013.

[Yen71] Yen, J. Finding the k shortest loopless paths in a network. Management Science, vol. 17, no. 11, pp. 712–716, 1971.

[YJF98] Yi, B-K., Jagadish, H., Faloutsos, C. Efficient retrieval of similar time sequences under time warping. In Proceedings of the IEEE International Conference on Data Engineering (ICDE), 1998.

[YZXS13] J. Yuan, Y. Zheng, X. Xie, and G. Sun. 2013a. T-Drive: Enhancing driving directions with taxi drivers. In Proceedings of the IEEE Transactions on Knowledge and Data Engineering (TKDE), 25(1), pp. 220–232, 2013.

[ZCFS+10] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I. Spark: Cluster computing with working sets. In Hot Cloud, 2010.

[ZYG12] Zhang, J., You, S., Gruenwald, L. U2STRA: High-performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs. In Proceedings of the ACM Workshop on City Data Management Workshop, (CDMW), pp. 5–12, 2012.

[ZYG13] Zhang, J., You, S., Gruenwald, L. GPU-based Spatial Indexing and Query Processing Using R-Trees. In ACM SIGSPATIAL Special, 6(3), pp. 23–31, 2013.

[Zheng15] Zheng, Y. Trajectory Data Mining: An Overview. ACM Transactions in Intelligent Systems and Technology (TIST), 6(3), 2015.

[ZXM10] Zheng, Y., Xie, X., Ma, W.-Y. Geolife: A collaborative social networking service among user, location and trajectory. IEEE Database Engineering Bulletin, 2010.

[ZZXZ12] Zheng, K., Zheng, Y., Xie, X., Zhou, X. Reducing Uncertainty of Low-Sampling Rate Trajectories. In Proceedings of the International Conference on Data Engineering (ICDE), 2012.

[ZZ11] Zheng, Y., Zhou, X (Eds). Computing with Spatial Trajectories. Springer, 2011.