Scalable GPU Algorithms for Similarity Query Processing and Clustering on Trajectory Data

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Hamza Mustafa

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Eleazar Leal

November 2019

Acknowledgements

I would like to thank Dr. Eleazar Leal for the amazing support he has provided throughout my Master's. I am grateful for his mentorship, his enthusiasm for teaching, his aptitude for research, and the countless hours he took out of his schedule to advise me. I would also like to thank Dr. Pete Willemsen for his valuable guidance and help. I appreciate all the knowledge he has given me, in his courses and beyond. Additionally, I would like to thank everyone in the Computer Science department for shaping my time at the University of Minnesota Duluth into an unforgettable experience.

Dedication

I would like to dedicate this thesis to my parents, Muhammad and Shazia Mustafa, for cheering me on from the sidelines. I'm grateful for their incredible support and love over the years.

Abstract

With the increasing prevalence of location sensor devices like GPS, it has been possible to collect large datasets of a special type of spatio-temporal data called trajectory data. A trajectory is a discrete sequence of positions that a moving object occupies in space as time passes. Such large datasets enable researchers to study the behavior of the objects describing these movements by issuing spatial queries such as trajectory similarity queries and trajectory clustering.

Top-K trajectory similarity queries retrieve the K most similar trajectories to a given query trajectory. This query has applications in many areas, such as urban planning, ecology and social networking; however, this query is computationally expensive. In this work, we introduce a new parallel top-K trajectory similarity query technique for GPUs, FastTopK, to deal with these challenges. Our experiments on two large real-life datasets showed that FastTopK produces on average 107.96X smaller candidate result sets, and 3.36X faster query execution times than the existing state-of-the-art technique, TKSimGPU.

The second type of trajectory query covered in this work is trajectory clustering. One important algorithm for clustering is DBSCAN, which is especially useful for finding clusters of arbitrary shapes. As opposed to other clustering techniques, like K-means, it does not require the number of clusters to be specified as an input parameter, and it is highly robust to outliers. However, DBSCAN has a worst-case quadratic time complexity that makes it difficult to handle large dataset sizes. To address this problem, several works have been proposed that exploit the massive parallelism of GPUs for DBSCAN clustering of point data. Nonetheless, none of these works have been experimentally compared against each other and none have been extended to cluster trajectory data. In this thesis, we review the existing GPU

algorithms for DBSCAN clustering on point data and conduct the first experimental study comparing these GPU algorithms using three real-world datasets to identify the best performing algorithm. Our results show that G-DBSCAN is the fastest being up to 969X faster than CPU DBSCAN on 128K points, while CUDA-DClust is the best performing GPU algorithm in terms of execution time and memory requirements, performing 53X faster than CPU DBSCAN while taking up to 166X less memory than G-DBSCAN.

Lastly, we use the work of our analysis of all the existing GPU-based DBSCAN clustering algorithms on point data to develop a new GPU-based trajectory clustering algorithm, GTRACLUS. Our experiments show that on two real world datasets, trajectory clustering can be made more than 20X faster than the CPU-based TRACLUS by using the proposed GTRACLUS algorithm.

Contents

List of Tables

List of Figures

1 Introduction

In this chapter, we present a brief overview of trajectories and GPUs, introduce some of the problems that exist in mining of trajectory data and present a proposed solution to these problems. We introduce the objective of the thesis in Section 1.1 and briefly provide reasons to employ GPU co-processors to run operations on trajectories. Section 1.2 presents the background on two types of trajectory queries that we focus on in this work, trajectory similarity query and trajectory clustering. It provides a description of GPU architecture and the research challenges it pertains. It also provides an introduction to trajectories, Hausdorff distance, similarity queries on trajectory data and the its research challenges. We briefly describe cluster analysis on trajectories and its applications, and present our contributions in consolidating all contemporary DBSCAN clustering algorithms in one comparative analysis. Moreover, we utilize those findings to present a new technique and compare it against the best trajectory clustering algorithm. Section 1.3 summarizes our contributions and finally, Section 1.4 provides a brief outline of the subsequent chapters.

1.1 Objective

A *trajectory* is a discrete sequence of positions that a moving object occupies in space as time goes by. Such trajectories can be collected through the use of location sensors like GPS by periodically sampling the movements of the objects and then concatenating these samples in a time-ordered sequence. Figure 1.1 shows an example

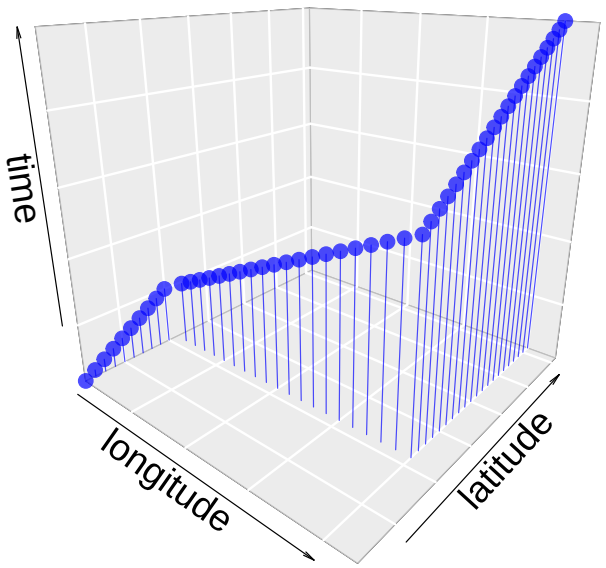

Figure 1.1: A location trajectory plotted on a 3D graph.

of a location trajectory plotted in 3D over longitude, latitude and time.

With the prevalence of increasingly accurate location acquisition devices, paired with advances in mobile computing techniques, it has been possible to collect large amounts of trajectory data, and the volume that is being accumulated of this type of data grows every second. Trajectory data provide an opportunity to understand the behavior of the associated moving objects by examining their previous, current and future location points. Querying these data can provide us with valuable insights which could potentially have far reaching applications in areas such as urban planning [67], ecology [26] [31] and social networking [64]. For example, Microsoft Research collected a broad range of user data for activities including hiking, cycling, sightseeing and shopping. This dataset, called the Geolife dataset [68], can be queried for applications such as location recommendations, user activity recognition, mobility pattern mining and location-based social networks. An example query of a location-based social network can be "based on a particular user's movement data, find 5 people

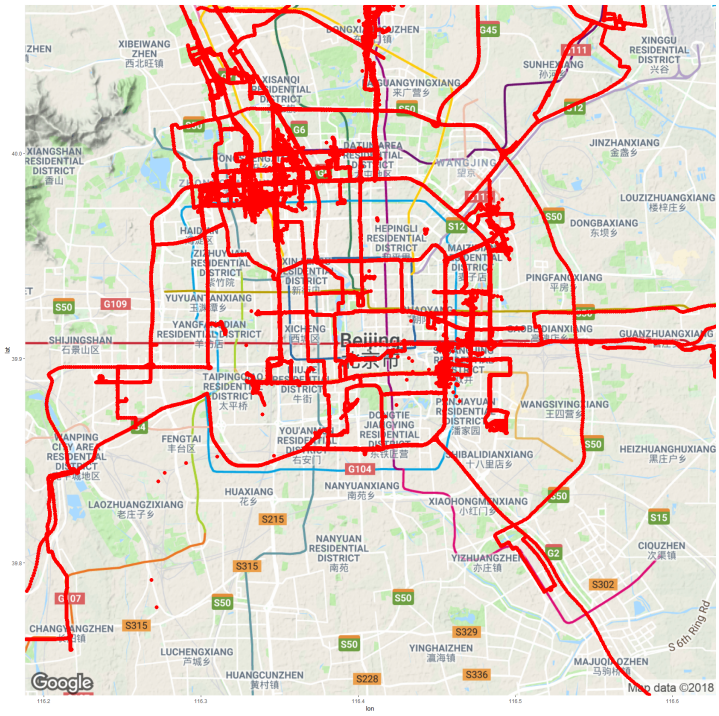

Figure 1.2: Trajectories in the Geolife dataset [68] plotted in red over a map of Beijing, China.

who frequent this biking trail or a trail most similar to it." Similarly, clustering operations can be performed on the trajectory data to gain useful insights, like finding objects that have moved in a similar pattern. An example would be "classify all the trajectories of hurricanes that took place in the last 15 years on the coast of (city) and correlate them with the damage caused to identify vulnerable regions in case of new hurricanes."

Because of its structure as a time-ordered sequence, trajectory data are more complex than point data, which makes processing it more difficult. To deal with this challenge, one can use parallel computer architectures like Graphics Processing Units (GPUs). These co-processors have many advantages: they are installed in all kinds of computers, from cellphones to supercomputers, are highly energy efficient [57], and can, under certain workloads, achieve up to an order of magnitude of higher

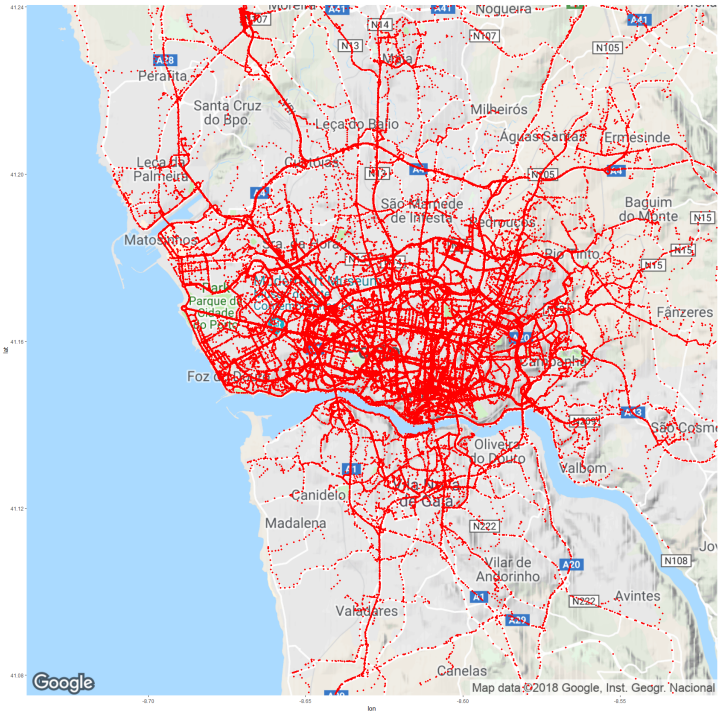

Figure 1.3: Trajectories in the Porto Taxi Service dataset [44] plotted in red over a map of Porto, Portugal.

instruction throughput than comparable multicore CPUs [38]. These characteristics of GPUs, in addition to the fact that it has been shown that it is possible to parallelize the similarity computation between any two trajectories [62], makes GPUs an ideal architecture on which top-K trajectory similarity queries or trajectory clustering can be processed.

This thesis explores the use of GPU-based algorithms for mining trajectory data. We discuss two novel algorithms: FastTopK, for similarity query processing, and GTRACLUS, for trajectory clustering. Moreover, we provide an experimental evaluation of GPU-based DBSCAN clustering algorithms and discuss the strengths and weaknesses of each.

1.2 Background

In this section, we present details on GPUs, clustering and DBSCAN as well as definitions of trajectory, Hausdorff distance, top-K trajectory similarity queries and GPU challenges. These concepts are mentioned throughout the remainder of this thesis.

1.2.1 Architecture

GPUs

GPUs are co-processors installed in most computers, from mobile phones to supercomputers, whose main purpose is the rendering of computer graphics. However, it is also possible to exploit the large parallelism potential of these co-processors to tackle large problems [7].

In this thesis, we use the vocabulary of CUDA C [48] to describe the programming model of GPUs. GPUs run code procedures called *kernels*, which look very similarly to regular C-language functions and are called from within the CPU execution flow. Because GPUs have their own separate memory address space, called *global memory*, the data that needs to be processed by a GPU is usually sent at the beginning of processing from the host's memory to the GPU's RAM, and then once the GPU finishes computing, the results are sent back from the GPU to the host's RAM.

GPUs can simultaneously run many threads. In order to manage this complexity, threads are logically grouped into GPU blocks. Each GPU block runs within a multiprocessor in the GPU, and as a result of this, the threads within each GPU block can communicate with each other through the GPU's *shared memory*, which is faster and smaller than global memory. The latest Nvidia GPU architecture, Turing, has 96 KBs of shared memory while the global memory goes up to 48 GBs in the Quadro

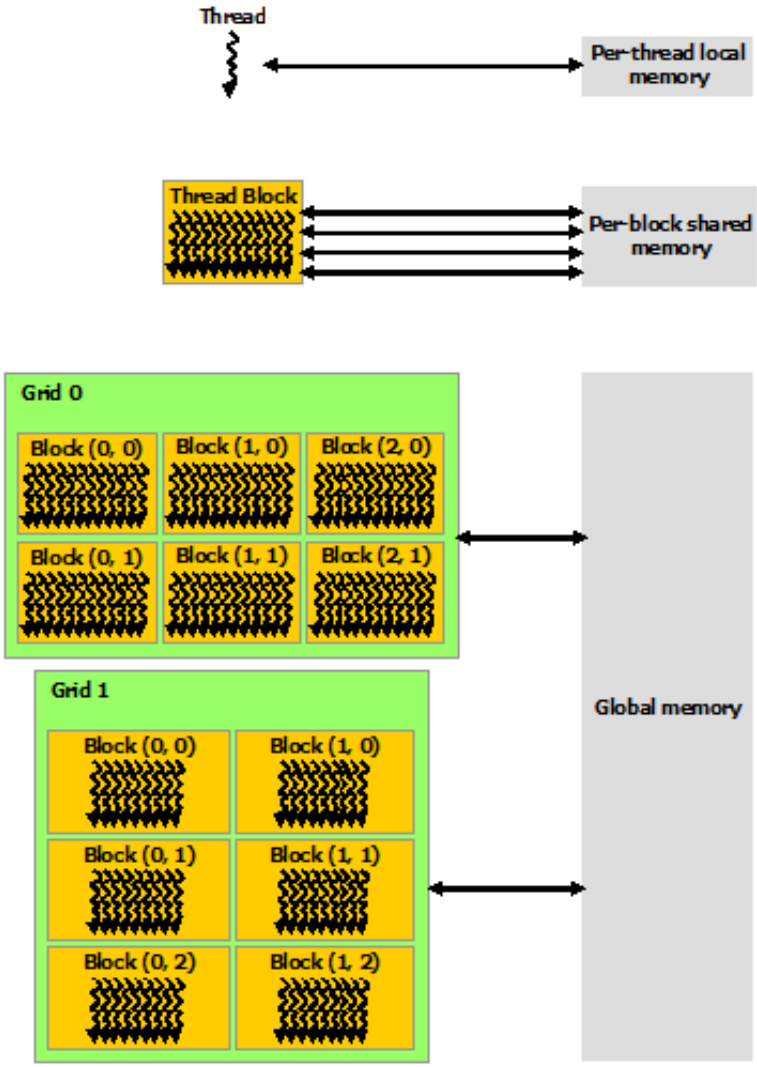

Figure 1.4: CUDA memory hierarchy (taken from docs.nvidia.com [48]).

RTX 8000 [49].

Figure 1.4 shows the memory hierarchy in CUDA. It shows a thread, which runs one instance of a kernel, accessing local memory available to it. A thread block, which is a group of threads, has shared memory available to all threads in the block which can be utilized for communication within a block. For sharing memory between blocks, global memory is used which is accessible to the entire GPU execution units i.e. threads, blocks, and grids. Figure 1.5 shows heterogeneous processing using CPUs

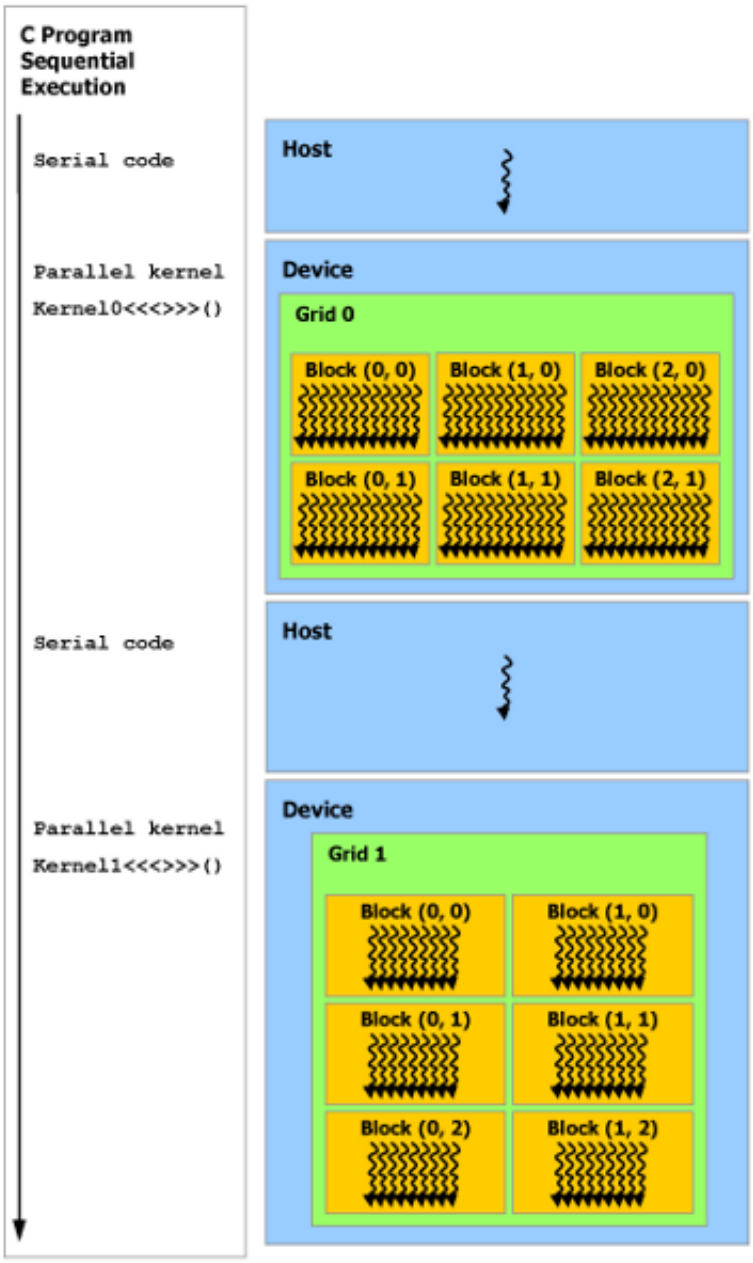

Figure 1.5: Serial and parallel code execution (taken from docs.nvidia.com [48]).

and GPUs, which is a very common form of algorithm execution. In this figure, the user is utilizing the strengths of both these architectures by running serial code on the host (CPU) and parallel code on the device (GPU). The device and host can work independently to process data using CUDA APIs. In Figure 1.5, after executing

`Serial code` on the CPU, the program calls `Kernel 0` which runs on the GPU. The results of this execution is used by another `Serial code` on CPU which executes `Kernel 1` for parallel execution again. The utilization of the two architectures on host and device makes this a heterogeneous process.

Research Issues of GPU programming

Due to the bandwidth-focused Single Program Multiple Data (SPMD) based architecture of GPUs, and the fact that they are co-processors with their own memory, all techniques that run on them face several challenges. Here we mention some of them:

1. One of the techniques we used to maximize the performance of our algorithm is memory coalescing. There are different memory spaces on a GPU device, the largest of which is the device DRAM. This memory is also called *global memory* because it is global in scope for all the blocks and the threads within them, as well as accessible and modifiable by the host. The device can access the global memory via 32, 64 and 128 byte transactions. A performance penalty is incurred if every thread generates a single memory access for 4 bytes for example. This is a potential issue for algorithms accessing non-contiguous memory addresses. So, to obtain maximum bandwidth and performance, it must be ensured that the data required by the kernel running on each thread is aligned and contiguous in the global memory. This practice is called *memory coalescing* [12].

2. Device DRAM is the only way of accessing data from within kernels running on GPU threads. This is why every data element to be processed by CUDA needs to be transferred from host (CPU) to device (GPU) memory. This poses a potential problem for algorithms reliant on lower bandwidth and lower latency,

that is, algorithms that require transfer of small amounts of data and high speed processing on it, since block-based memory copy from device to host takes practically constant amount of time. This means that GPU-based executions of algorithms which rely on a asymptotic time proportional to the data size do not provide significant speedup if the data are not large to begin with, since in this case the transfer time overhead dominates the processing time. Several techniques exist to circumvent this, for example, converting stream processing algorithms to batch processing in order to collect adequate amount of data to make GPU processing beneficial.

3. Running code on GPU requires initiation by means of a kernel call, as shown in Figure 1.5. Every call to the kernel has an overhead because of driver instructions that tell the device to execute CUDA code. This overhead adds up when multiple kernels are called, which makes the program execute less efficiently. One way to overcome this is to combine the operation of multiple kernels into one kernel and to call this combined kernel once. This approach has limitations in cases where the output from the first kernel needs to be processed by the host before providing it as input to the subsequent kernel. Another way to overcome kernel launch overhead is to utilize the fact that kernel calls are asynchronous. This means that the host does not need to wait for kernel execution to complete to execute the instructions on CPU that come after the kernel call in the program. This approach requires the program to delay using `cudaMemcpyDeviceToHost` for as long as the kernel is executing because this function essentially makes the kernel call synchronous by making the host wait until kernel execution completes.

4. The CUDA architecture is built around a scalable array of multithreaded Stream-

ing Multiprocessors (SMs). These SMs are Single Program Multiple Data (SPMD) architectures, which schedule the same instructions on multiple threads but with different data for each thread. Branching instructions (like `if` blocks running different code) pose a problem at the architectural level since all threads are not running the same instructions. This causes a delay in the program execution. Barrier synchronization commands like `syncthreads()` can be used up to a certain extent to make sure all the threads remain synchronized.

5. Streaming Multiprocessors (SMs) create, manage, schedule, and execute threads in groups of 32 parallel threads called warps [12]. *Occupancy* is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. Low occupancy means that the processing resources available on the device are not being fully utilized, which results in poor efficiency. However, any more than the maximum supported occupancy on a particular hardware also results in performance degradation because of reduction in resources per thread. Thus, in order to make the best use of a CUDA device, occupancy must be maximized. Occupancy is hardware-dependent and can be increased by adjusting the size and number of thread blocks, as well as thread memory and shared memory [12].

6. Sequential processes are programs that have dependency on the output of previous stage in processing [54]. These processes are difficult to parallelize since they don't apply instructions on different data in parallel. However, several works have been proposed that circumvent these issues for specific applications, like exclusive scan [61].

1.2.2 Trajectories

Here we present the formal definition of trajectories, the sequence of points occupied by an object in multi-dimensional space as time goes by.

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

Figure 1.1 shows a 3D representation of a trajectory. It uses two spatial dimensions and one time axis to plot the multi-variable time series data.

As mentioned in Section 1.1 and detailed in the following chapters, applying the same operations as point data on trajectory data is not straightforward. The fact that every single entity in trajectory data is a sequence of multi-variable points poses new challenges in terms of correctness and computational complexity among others. This is a key concept that is thoroughly detailed in this work as we develop new techniques to search for similar trajectories and group trajectories with unsupervised learning.

1.2.3 Trajectory Similarity Queries

Hausdorff Distance

Here we present the formal definition of Hausdorff distance used in Chapter 2. The Hausdorff distance is a *metric* that tells the furthest distance between two sets of points, such that if a distance is calculated from a random point in one set to its closest point in the other set, this distance will be less than or equal to the Hausdorff distance between the two sets.

Given two trajectories P and Q, the *Hausdorff distance hausd(P,Q)* between them

is defined as the maximum between $max_{p \in P} min_{q \in Q} d(p, q)$ and $max_{q \in Q} min_{p \in P} d(p, q)$, where *(p,q)* is the Euclidean distance between the points p and q [47]. Figure 1.6 presents an example of Hausdorff distance calculation between two trajectories.

A *metric* is a distance measure such that it follows three properties: the distance of a point to itself is zero, the distance is always non-negative, and the distance between three points follows the triangle inequality.

Top-K Trajectory Similarity Queries

In Section 1.1, we provided an example of a top-K trajectory similarity query which was, "based on a particular user's movement data, find 5 people who frequent this biking trail or a trail most similar to it". For a given positive integer K, a trajectory dataset Q and a query trajectory P, a top-K similarity query provides to the user K trajectories from Q which are closest to P. Another example of this would be "given the migration trajectory of sparrows, find 3 other bird species with the most similar migration trajectories to those of the sparrow." This particular query finds its application in ecology [26] [31]. Top-K trajectory similarity queries are useful in many application domains because end-users are more interested in the most relevant answers contained in a potentially huge answer space. Following is the formal definition of a top-K trajectory similarity query.

Given a positive integer $K > 0$, two finite non-empty sets of trajectories P (the query set) and Q (the database), and a similarity measure $\sigma : P \times Q \rightarrow \mathbb{R}^3$, a top-K trajectory similarity query returns for every p in P a set $R_p \subseteq Q$ satisfying that $|R_p| = K$ and for every $q \subseteq R_p$ and $q_{other} \subseteq Q - R_p$ it is the case that $\sigma(p, q_{other}) \leq \sigma(p, q)$.

Figure 1.6 shows us a similarity query taking place between two trajectories and their Hausdorff distance being calculated. Both the query trajectory and the database

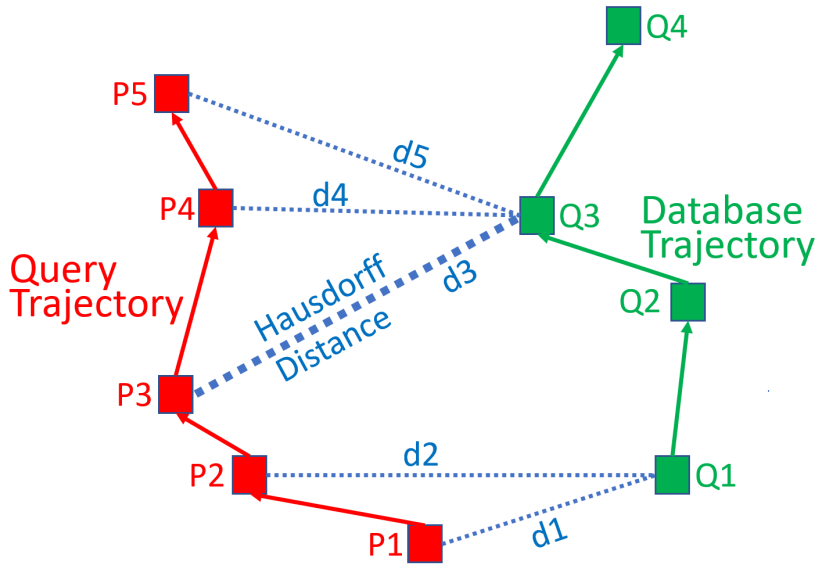

Figure 1.6: Finding similar query through multiple Euclidean distance calculations.

trajectory need not have the same number of points, here the query trajectory has five points: P1, P2, P3, P4, and P5, while the database trajectory has four: Q1, Q2, Q3, and Q4. Starting off from its first point P1, the query trajectory evaluates the distance to all points in the database trajectory and picks the smallest distance d1. d1 is the distance between the closest point Q1 in database trajectory to the query trajectory point P1. Similarly, all points in the query trajectory pick a minimum distance out of all the distances calculated, as plotted with dotted lines and labeled d1-d5 in Figure 1.6. Out of all the minimum distances, d1 to d5, the Hausdorff distance is the maximum distance among them. In Figure 1.6 the maximum distance is d3 which is labeled as the Hausdorff distance.

Research Challenges

The problem of top-K trajectory similarity query processing is challenging because of several reasons:

1. The datasets involved are often very large, and with the ubiquity of location sensors, the sizes of these datasets will keep growing.

2. The trajectories belonging to these datasets are complex objects, each of which consists of potentially many points in space. Each of these points can contain, besides a latitude, longitude and timestamp, many other attributes.

3. The time complexity of computing the similarity between any two trajectories is, in general $O(n^2)$ where n is the number of points in all the trajectories, which makes it a computationally expensive procedure.

GeoLife [68] is an example of a dataset that demonstrates the challenges mentioned previously. The trajectories of this dataset are plotted in red in Figure 1.2. GeoLife has 17,600+ trajectories spanning across 1.2 million kilometers, and a total duration of 48,000+ hours. Using a naïve algorithm to find the K trajectories in GeoLife that are the most similar to a given query trajectory could potentially entail performing up to 536 trillion Euclidean distance computations. The Taxi Service dataset, plotted in Figure 1.3, is also another example of a large trajectory dataset with 1,700,000+ trajectories [44]. This dataset consists of the trajectories described by 442 taxis running for one year in the city of Porto, Portugal.

Summary of the Proposed Solution

Despite that GPUs have been shown to offer large performance improvements over CPUs [7], there are very few top-K trajectory similarity query processing techniques that exploit the potential of GPUs. The most recent of them is TKSimGPU [35] [36], which is a filter-and-refine algorithm. As such, this algorithm first generates a candidate result set with fewer trajectories than the entire database, and then

performs an exhaustive search on this smaller candidate result set. TKSimGPU has two disadvantages:

- The sizes of the candidate result sets generated are still considerably large, which leads to longer query execution times.
- This algorithm requires two input parameters that must be defined by the user: the size of the grid and the sample size. The inconvenience with having more user-defined input parameters is that in general it makes it harder to find consistently good performance.

In this work, we introduce a new top-K trajectory similarity query processing algorithm, FastTopK, that employs a simpler pruning strategy, which produces smaller candidate result sets, and thus, has faster query execution times than TKSimGPU. FastTopK also has the advantage that it requires only one user-defined input parameter, the grid size, unlike TKSimGPU, which requires two user-defined parameters.

1.2.4 Trajectory Clustering

Cluster analysis is a data mining task that consists in that given a dataset D and a similarity measure M, find a set of k subsets of D such that the points within each of these subsets are highly similar to each other (cohesion), while also guaranteeing that any two points p and q belonging to different subsets are as dissimilar to each other as possible (separation). In some algorithms, the value of k is provided as an input parameter to the clustering method, while in others, the algorithm itself determines the value of k.

Detection of clusters is one of the most common unsupervised learning techniques used in data mining to find the relationship between data points in close proximity

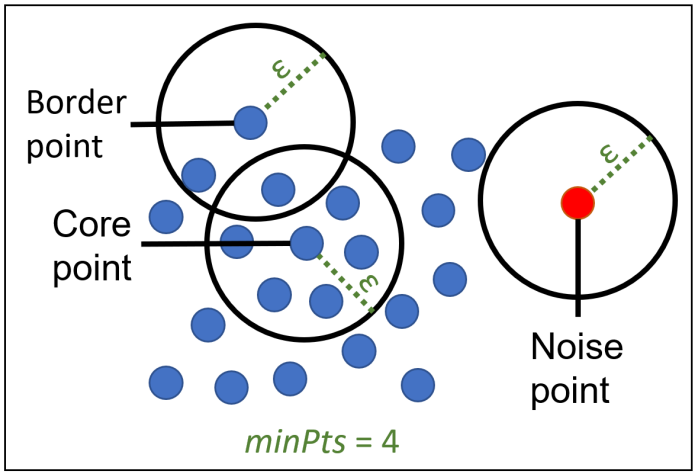

Figure 1.7: Classification of core point and border point using DBSCAN.

to each other with regards to a similarity measure. It is commonly used in statistical data analysis and in several fields including machine learning, computer graphics, image analysis and pattern recognition [22]. Cluster detection can also be used as a pre-processing method for supervised learning methods.

Clustering can be categorized as hard or soft. *Hard clustering* is when each data point belongs to a cluster or it does not. *Soft clustering* is when all data points have certain degree of membership to a class. We also discuss works that touch briefly on clustering techniques such as Hierarchical clustering, Centroid-based clustering (K-Means), Distribution-based clustering (EM algorithm), and Density-based clustering (DBSCAN). Chapter 3 is dedicated to the evaluation of several GPU-based DBSCAN technique and Chapter 4 contains its application of grouping trajectories. Since our major focus is on this particular clustering technique, we introduce it in more detail below.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [13] is a density-based clustering that has the following advantages:

- It can form clusters of arbitrary shapes.

- It is robust to noise.
- It does not need the number of clusters to be given an input parameter.

Figure 1.7 shows an example of classification of point data using DBSCAN. DBSCAN takes two parameters as input, *epsilon* and *minPts*. *epsilon* determines the radius of the circle that marks the *epsilon-neighborhood* of a point. In Figure 1.7 the *epsilon* is ϵ. The number of points that lie inside the *epsilon-neighborhood* of every point is determined and compared with *minPts*. In Figure 1.7 *minPts* is 4 and we can see that the core point has 6 points in its *epsilon-neighborhood*, which makes it a core-point of a cluster. Similarly, all points that have 4 or more points in their *epsilon-neighborhood* are core-points. The cluster formed by these core points is colored in blue. However, there are some points like the border point in Figure 1.7 that are still part of the cluster but do not have *minPts* number of points in its *epsilon-neighborhood*. These points have at least one core point in their *epsilon-neighborhood*. Finally, any point that is neither core nor border is labeled as noise as shown in red in Figure 1.7.

DBSCAN clustering algorithm is extended in TRACLUS [37] to perform clustering on multi-variable time-series data. Trajectory clustering is one of the two main operations performed by TRACLUS and the complexity of DBSCAN increases by the length of trajectories which makes it even more computationally intensive than when run on point data. This clustering of trajectories however, is a vital tool in determining the similarities between the time series data and its analysis. For example, clustering the animal movements near roads may provide useful insights on the impact of motorized vehicles to their natural habitat. In addition to clustering, a major role of TRACLUS is to partition the trajectories before clustering. This partitioning is done using the MDL principle and is used to find common sub-trajectories among

large trajectory data.

Research Challenges

The research challenges of clustering are as follows:

- The continuous increase in the sizes of datasets means that a technique like DBSCAN, which has a worst-case time complexity of $O(n^2)$ [28], may not scale for these large dataset sizes.
- The reason for this worst-case time complexity of DBSCAN lies in the fact that it may potentially require computing the distances between every two points in a dataset.

Summary of the Proposed Solution

This scalability issue of DBSCAN can be addressed, in part, through the use of spatial data structures like R-trees [20], which can reduce the number of distance calculations. Another approach, which does not exclude the use of R-trees, to cope with this scalability issue consists in taking advantage of parallel computing architectures like multicore CPUs and GPUs. Both these architectures share in common that they are readily available in all kinds of computers, from mobile phones to supercomputers. GPUs, however, can have up to an order of magnitude of higher instruction throughput than comparable multicore CPUs [38]; and GPUs are very energy efficient [27]. All these advantages make GPUs an ideal parallel architecture that can be used to tackle the scalability issues of DBSCAN.

Previous research has indeed focused on how GPUs can be exploited for DBSCAN [3] [59] [6], and all these existing works have claimed significant execution time speedups over the original (serial) DBSCAN algorithm [13]. However, choosing

among them for a specific application is problematic because no benchmarking has been done that compares the performance of these algorithms against each other. In each of these works, the GPU-based DBSCAN algorithms proposed have been compared only against the original CPU algorithm [13] with each paper using a different dataset. Moreover, the disparities between the years of publication have resulted in these experiments running on substantially different hardware, which makes the task of choosing the best GPU algorithm even more difficult.

In this thesis we review the existing GPU-based DBSCAN algorithms and analyze the techniques they incorporate to achieve desired results. We also run these algorithms on the same hardware with three real-world datasets to evaluate the strengths of each and find their efficiency in terms of processing time and memory consumption.

From our experimental results, we find G-DBSCAN technique to be the fastest among all the GPU-based DBSCAN clustering algorithms. So we utilize this data to alleviate the shortcomings of the existing trajectory clustering algorithm, TRACLUS. For real world data, TRACLUS requires a high amount of computational time due to MDL calculations and trajectory points comparison against every other dataset point. In our tests we found out that for 100k trajectories TRACLUS takes about forty minutes. Employing known parallel computational strategies for the partitioning stage and an extended, trajectory analyzing version of GDBSCAN in the grouping stage, we create GTRACLUS, a GPU-based trajectory partitioning and clustering framework. In our experiments, GTRACLUS performed more than 20X faster than TRACLUS algorithm for trajectory clustering and 10X faster for trajectory partitioning bringing the total time for 100k trajectories clustering down to 2.2 minutes.

1.3 Contributions

The contributions of this thesis are the following:

- We propose a new parallel top-K trajectory similarity query processing algorithm for GPUs, FastTopK [46], that produces smaller candidate result sets than the state-of-the-art technique, TKSimGPU [35], and therefore has faster query execution times. FastTopK also has the advantage that it requires only one user-defined input parameter, the grid size, unlike TKSimGPU, which requires two user-defined input parameters.

- We perform an experimental comparison of our proposed technique [46] against TKSimGPU using two real-life large trajectory datasets, GeoLife and Taxi Service Trajectory Prediction.

- We review the existing DBSCAN algorithms that have been specifically designed for GPUs.

- We present the first experimental study [45] comparing the existing GPU algorithms for DBSCAN clustering using three real-world datasets; the purpose of this experimental comparison is to identify the best-performing DBSCAN clustering technique for GPUs in terms of execution time and memory requirements.

- We utilize the results of our study to create GTRACLUS, which partitions trajectories using GPUs and then uses a modified version of G-DBSCAN to cluster them using a graph and breadth first search approach. Our experiments show that GTRACLUS is more than 20X faster than the original TRACLUS algorithm, running experiments using a GPU in less than 3 minutes that take 40 minutes on a multi-core CPU with four threads.

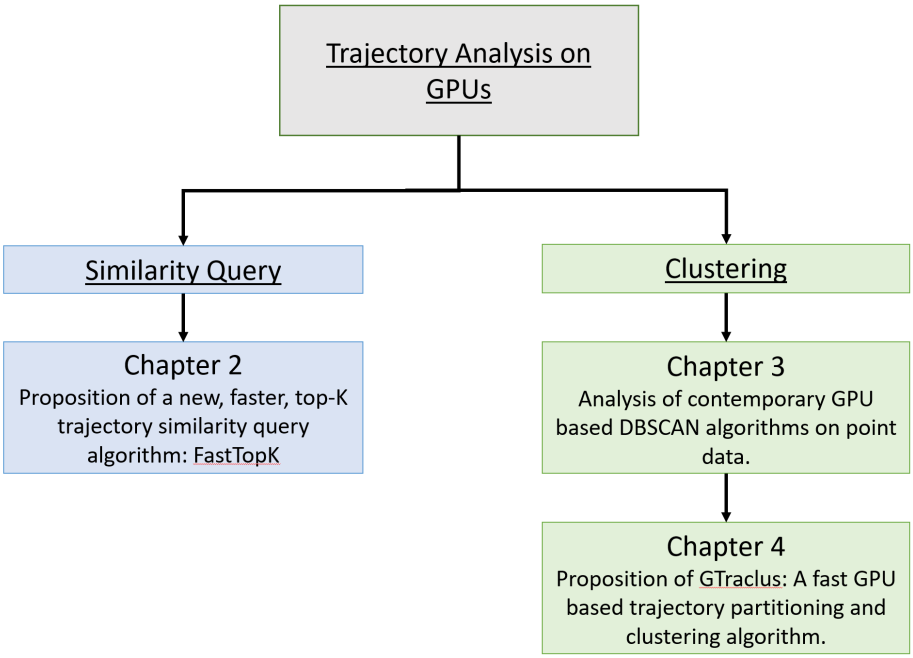

Figure 1.8: Outline of the material discussed in this work.

1.4 Outline

The rest of the thesis is organized as follows and illustrated in Figure 1.8. Chapter 2 discusses the top-K trajectory query algorithm, FastTopK, in detail with related work and experimental results. Chapter 3 presents a survey of the existing GPU algorithms for DBSCAN clustering and an extensive experimental evaluation of all the algorithms. Chapter 4 is a logical continuation of the results from the DBSCAN experiments to propose GTRACLUS, a faster trajectory partitioning and clustering algorithm that utilizes GPUs for its execution. Finally, Chapter 5 offers conclusions and future work.

2 FastTopK: Trajectory Similarity Queries with GPUs

In this chapter, we present FastTopK [46]: a novel, GPU-based, trajectory search algorithm that requires fewer parameters and does more efficient pruning than any other GPU-based trajectory search algorithm. In Section 2.1 we present related work. Section 2.2 describes the proposed algorithm in detail and Section 2.3 and 2.4 provide experimental evaluation of FastTopK. Finally Section 2.5 presents discussion.

2.1 Related Work

There exist several works that have proposed serial algorithms for processing trajectory similarity queries. Among them we find ERP [8] , EDR [9], wDF [11], KSQ [42], and EDwP [53]. Unlike these techniques, our proposed algorithm uses the Hausdorff distance [47]. This distance arises naturally in problems related to urban planning, such as when changing bus routes [47]. This distance measures the longest any person would need to walk from their usual stop in the old route to the nearest stop in the new one. The Hausdorff distance has another advantage over EDR and EDwP, which is that it is a metric, and for this reason, it allows the use of the triangular inequality in order to do pruning.

Another difference between all the works mentioned before and ours is that these techniques were designed for running on single cores. However, to obtain scalable

algorithms that run on parallel architectures, such as GPUs and multicore CPUs, they must be designed in such a way so as to exploit the parallelism and cater to the issues of these architectures, some of which such as the memory coalescing of global memory accesses on GPUs are explained in Chapter 1. FastTopK is an algorithm that is specifically designed for parallel GPU processing.

There are far fewer existing parallel algorithms for trajectory similarity query processing than there are for single core architectures. U2STRA [63] is a GPU algorithm that also uses the Hausdorff distance for performing near-join trajectory similarity queries. The query that it executes receives as inputs a positive real number epsilon, a query trajectory, and a database of trajectories, and outputs all the trajectories in the database whose similarity with the query trajectory is at least epsilon.

The technique [1] [19] also processes near-join similarity queries, but uses the Euclidean distance, instead of the Hausdorff distance, and applies them to the study of moving galaxies.

Another parallel technique for trajectory similarity query processing on GPUs is TKSimGPU [35] [36]. This algorithm, unlike U2STRA, is designed for Top-K trajectory similarity query processing. The key idea behind this technique is to draw a sample of trajectories in the database, and then compute the average Hausdorff similarity between the query trajectories and the sampled database trajectories. This average is called epsilon. Then all the trajectories are rasterized by laying a grid on top of the space in which trajectories lie. Then the Minimum Bounding Rectangles (MBRs) are computed for all query and database trajectories. However, the MBRs of the database trajectories are expanded by an epsilon amount in all directions, so that if a query trajectory's MBR intersects with the expanded MBR of a database trajectory, then the database trajectory in question is added to the candidate result set.

A disadvantage of TKSimGPU is that, despite that it greatly reduces the sizes of the candidate result sets obtained, it can still produce large sets, and this leads to both longer query execution times, and higher memory consumption in an already memory constrained architecture as GPUs. To address this issue, FastTopK does not perform any sampling of the database trajectories, and it does not rasterize trajectories using MBRs. Instead, it rasterizes trajectories directly using grid cells. These two differences lead FastTopK to producing smaller candidate result sets than TKSimGPU.

2.2 Proposed Algorithm

In this section, we present a new parallel GPU-based algorithm for Top-K trajectory similarity query processing called FastTopK [46], whose pseudo-code is presented in Algorithm 2.1.

2.2.1 Overview

FastTopK receives as input parameters a positive integer K, a query trajectory, and a database of trajectories. As output, it returns the K trajectories in the database that are the most similar to the query trajectory.

FastTopK consists of three stages: the set-up stage, the filter stage, and the refine stage. In the set-up stage, we initialize a uniform grid of a fixed length, which serves as a way to classify all query and database trajectories. The filter stage extracts K or more trajectories closest to the query trajectory from the database in terms of distance. This stage is where the pruning takes place, which consists in discarding certain database trajectories that do not belong to the result set. The refine stage takes as input the remaining un-discarded database trajectories and uses GPU ker-

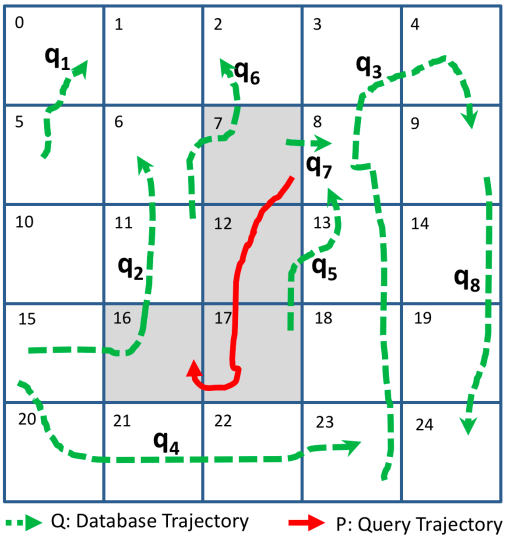

Figure 2.1: A sample 5x5 grid with database and query trajectories plotted. The grey area represents the query cells that result after rasterization.

nels to compute their Hausdorff distances to the query trajectory. These results are sorted with respect to their Hausdorff distances, and then the trajectories with the K shortest distances are returned.

2.2.2 First Stage: Set-up

The purpose of the first stage, the set-up stage, is to apply a uniform grid on the database. This uniform grid must span across the entire spatial extent of all trajectories in the database. An example illustrating this is shown in Figure 2.1, where the database consists of all the green, dashed trajectories, namely $q_1, q_2, \ldots, q_7$, and the query set consists of a single trajectory p, shown with a red and contiguous line. This same figure shows a uniform grid consisting of 25 cells. It can be seen in this example that no trajectory lies outside of the grid. This grid is an important part of the algorithm, as it partitions a trajectory into smaller portions, one corresponding to every grid cell, and this allows an easy parallelization of the query computations.

The grid initialization takes place at Line 2 of the algorithm in Algorithm 2.1. The initialization consists of creating `grid size` $\times$ `grid size` cells.

2.2.3 Second Stage: Filter

The filter stage takes place in Line 3 of Algorithm 2.1, which then calls the *filter* function on Line 8 of the same algorithm. The first step in this stage consists in rasterizing each trajectory in the *database* (Line 9). Rasterization consists in assigning to each trajectory the set of grid cells that it intersects. This assignment is performed on the GPU because this coprocessor can better exploit the parallel nature of this operation. We provide the grid size and the database trajectories to a kernel, which in turn determines all the grid cells where the trajectory lies based on its boundary points. This can be seen in Line 18 of the function *rasterize*. This function is implemented to maximize the GPU performance. The trajectory IDs and cell IDs are stored in contiguous memory blocks of the global GPU memory which allows memory coalesced global accesses for the threads. Each thread is assigned a trajectory to maximize the GPU occupancy. Every thread computes the boundary points for each trajectory and returns the intersecting cells with this MBR to the global GPU memory in a contiguous manner.

After rasterizing each trajectory, each grid cell is associated with a list of the trajectories contained in it. This takes place in Line 19. For example, for the 5x5 grid in Figure 2.1, after rasterization takes place, grid cell 8 is associated with the trajectories q_3, q_5 and q_7; this is because trajectories q_3, q_5 and q_7 intersect this grid cell. Similarly, grid cell 2 is associated with trajectory q_6.

Note that once the rasterization of the database is done, this process need not be repeated for different query trajectories or for different values of K, unless either the

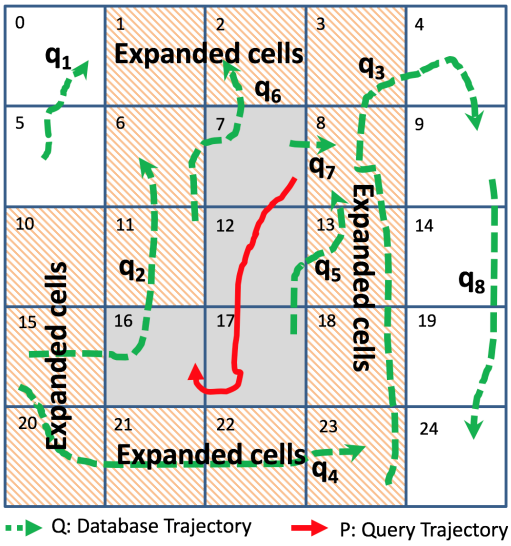

Figure 2.2: Grid after cell expansion.

database or the grid size changes. In other words, the database rasterization is done only once per database, irrespective of changes in either the query trajectory or of the value of K.

After rasterizing the trajectories in the database, FastTopK rasterizes the query trajectory (Line 10). For example, in Figure 2.1, the grid cells 7, 12, 16 and 17, which are grayed out, are the *query cells*. These are the grid cells intersected by the query trajectory. Then we find the intersection between the query grid cells and the database grid cells. This is done by *findOverlapTrajs* in Lines 21 to 24. The classified grid we obtained from rasterizing the database assists us in doing this. In our example from Figure 2.1, we observe that query cell 7 is associated with trajectories q_6, and q_7, grid cell 12 contains trajectory q_5, and grid cell 17 contains trajectory q_5 as well. The resulting candidate trajectory result set is q_2, q_5, q_5, q_6, and q_7. Duplicate results are obtained because any given trajectory can intersect multiple grid cells. Line 24 removes these duplicate results.

Algorithm 2.1: FastTopK Algorithm

Input: Trajectory query, Database of trajectories, K, GridSize

Output: The K database trajectories most similar to the query trajectory

```
1  function fastTopK(database,query,K,GridSize)
2      myGrid ←create a grid with GridSize x GridSize cells;
3      filtTraj ← filter(K, myGrid, database, query);
4      hausdorff ← findHausdorffDistance(results, query);
5      sortedTraj ←sort filtTraj with hausdorff as key;
6      topKResults ←sortedTraj{1, ..., K};
7      return topKResults;

8  function filter(K,myGrid,database,query)
9      databaseCells ← rasterize(myGrid, database);
10     queryCells ← rasterize(myGrid, query);
11     results ← findOverlapTrajs(queryCells, databaseCells);
12     while results.size() < K do
13         Expand queryCells in every direction by 1 cell;
14         results ← findOverlapTrajs(queryCells, databaseCells);
15     return results;

16 function rasterize(grid,trajectories)
17     for each trajectory in trajectories do in parallel
18         trajCells ←find all the cells in which the trajectory resides;
19         grid.cells[trajCells] ←trajectory;
20     return grid;

21 function findOverlapTrajs(queryCells,databaseCells)
22     for each queryCell in queryCells do
23         results ←databaseCells.cells[queryCell];
24     removeDuplicates(results);
25     return results;

26 function findHausdorffDistance(database,query)
27     for each trajectory in database do in parallel
28         hausdorff ←Hausdorff distance between query and database
             trajectory;
29     return hausdorff;

30 results ← fastTopK(database, query, K, gridSize);
```

After removing duplicates, the algorithm counts the number of trajectories in the candidate result set. If this number is greater than or equal to K, FastTopK proceeds to the next step. Otherwise, the algorithm performs query cell expansion, which will be now explained. In our example, we obtained four trajectories after removing duplicates: q_2, q_5, q_6, and q_7.

Let's assume for the rest of this example that K is 5. From Lines 12 to 14 the while loop ensures that the number of resulting trajectories is at least K. For this purpose, it expands the set of query cells by adding those grid cells that are immediately adjacent, either vertically, horizontally or diagonally, to the ones already in the set of query cells. This is illustrated in Figure 2.2, which shows the expanded cells with orange stripes and marked as "Expanded cells." In our example, the expansion of the query cells will add every cell to the immediate neighborhood of the current set of query cells: 7, 12, 16, 17 . This means that after query cell expansion, the set query cells will expand to include grid cells 1, 2, 3, 6, 7, 8, 10, 11, 12, 13, 15, 16, 17, 18, 20, 21, 22, 23 . These cells are checked to see if they intersect any database trajectories. In this case, after removing duplicates, the results come out to be trajectories q_2, q_3, q_4, q_5, q_6 and q_7. Since the size of this new candidate result set is larger than $K = 5$, FastTopK proceeds to the refine stage. If the candidate result set had not had at least $K = 5$ trajectories, then FastTopK would have performed query cell expansion again, but on the newer set of query cells.

2.2.4 Third Stage: Refine

Once the candidate result set is obtained, all that remains is to calculate the Hausdorff distance between each candidate trajectory and the query trajectory, then sort these distances and candidate pairs in increasing distance, and then output the K

candidates closest to the query. This is because the candidate result set is guaranteed to be a super set of the query result set.

Between Lines 26 and 28, the Hausdorff distance between the query trajectory and every trajectory in the candidate result set is calculated. Because of the inherent parallelism of the task, this is done using GPU kernels. Every CUDA block oversees the computation of the Hausdorff distance between the query trajectory and a single database trajectory. Every thread within a block is assigned exactly one point in the query trajectory, and exactly one database trajectory. Then each thread t calculates the set D_t of Euclidean distances between its assigned query trajectory point and all the points of its assigned database trajectory, and out of all these distances contained in D_t, each thread t will select the minimum distance m_t.

Once each thread t has computed its m_t, FastTopK performs a block max reduction [24] to find the maximum value of m_t over all t. Running this kernel twice with the roles of the query set and database reversed gives us the Hausdorff distance between the query trajectory and a given database trajectory.

After the execution of the Hausdorff distance calculation kernel (Line 4 of Algorithm 2.1), all the results are moved back from the device's global memory space to the host's. These distances are sorted along with their trajectory identifiers. (Line 5 of Algorithm 2.1). Once the trajectory identifiers and their Hausdorff distances are sorted in increasing order of their Hausdorff distances, the first K results are extracted and returned as output. This is performed in Lines 6 and 7 of Algorithm 2.1.

2.3 Performance Analysis

In this section, we describe the experiments performed to evaluate the performance of our proposed FastTopK algorithm [46].

2.3.1 Datasets

For our experiments, we used two real-life datasets: GeoLife [68] and the Taxi Service Trajectory Prediction Challenge dataset [44]. The GeoLife dataset, plotted in Figure 1.2, contains 17,621 trajectories captured by 182 users in a period of over three years. GeoLife was collected by Microsoft Research Asia and recorded using different GPS location sensors. The total distance of this dataset is about 1.2 million kilometers, and the total duration is over 48,000+ hours [68]. Every data point contains four values: longitude, latitude, altitude and time.

The Taxi Service Trajectory Prediction Challenge dataset [44] [50], shown in Figure 2.3, consists of the trajectories of 442 taxis running in the city of Porto, in Portugal, over the period of one year. It has 1,710,000+ trajectories with 81,000,000+ points in total. Every data point in this dataset has, besides longitude and latitude values, a unique identifier for each taxi trip, taxi ID, timestamp, and user information.

We have removed all the trajectories consisting of only one point from both datasets. We have also segmented the trajectories by doing stay-point detection [66], in which trajectories are split if the object in question remains stationary for longer than 30 minutes [35]. We also split the original trajectories into 256 point sub-trajectories [35], so that a single trajectory does not span across the entire map, which could hinder the filter stages of both algorithms. After these processing steps, 64,000 trajectories remain in the GeoLife dataset. In the case of the Taxi Service Trajectory dataset, we chose a subset of 1,700,000 trajectories for our experiments.

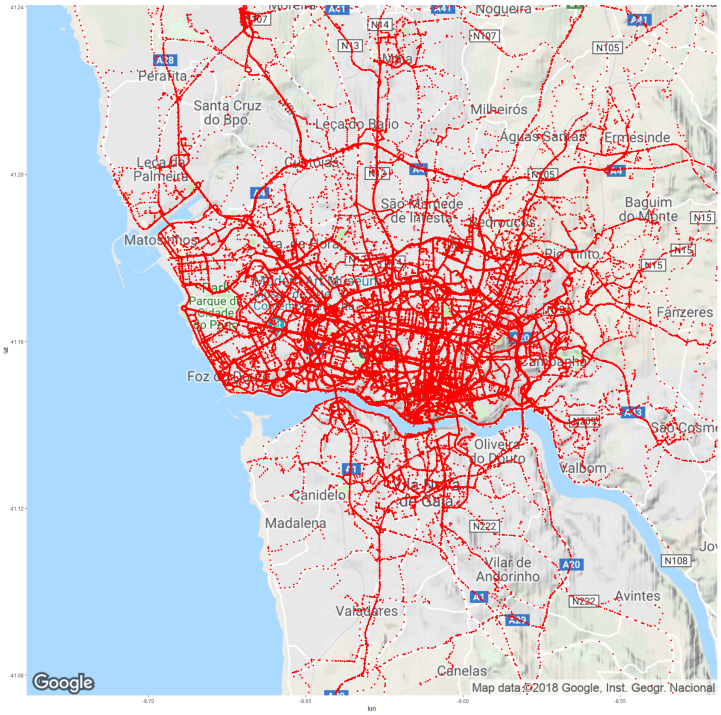

Figure 2.3: Trajectories in the Porto Taxi Service dataset plotted in red over map of Porto,Portugal.

In the case of GeoLife, we also removed all the trajectories that were not labeled as "walk"; this was done because the trajectories labeled as "walk" tend to be shorter and have smaller MBRs.

2.3.2 Hardware

All experiments were performed on a machine equipped with a four-core Intel Core i5 3470 chip running at 3.2 GHz, a GTX 1060 GPU with 6 GB of GDDR5 Memory, and a system RAM of 8 GB. The program was executed and benchmarked on Ubuntu 16.04 with Linux kernel version 4.15.0. Both algorithms were compiled with `O2` optimizations with CUDA 9.2 using Thrust v1.8.1 and CUB v1.8.0 [33].

2.3.3 Competing Techniques

In the following experiments, we compare our proposed GPU algorithm, FastTopK [46], with the only other existing Top-K trajectory query algorithm on GPUs, TKSimGPU [35]. According to the experiments performed on TKSimGPU, this algorithm is 3.8X faster than a multicore CPU implementation, and 3.37X faster than a naive GPU exhaustive search algorithm [35].

One advantage of FastTopK over TKSimGPU is that, unlike the latter algorithm, it does away with the computation of MBRs, effectively treating the boundaries of the cells in the grid as MBRs. This makes the algorithm simpler to implement and faster to execute due to the fewer operations required for pruning.

To keep the comparisons fair, the exact same GPU kernel `findHausdorffDistance` (Line 26 in Algorithm 2.1) is used in the refine stages of both algorithms.

2.3.4 Evaluation Metrics

We compare our proposed algorithm, FastTopK, against TKSimGPU in terms of the total query set execution time for a given database and query sets. In other words, the execution time is measured since the moment when the database and query sets are received, and is stopped at the point when all query trajectories in the query set Q have been executed. Each experiment is repeated five times, and then the average of these five executions is reported for each experiment.

Table 2.2 breaks down the total query set execution time into its components: the filter time and refine time. The filter time is the time it takes to prune the trajectories. The refine time is the time it takes to calculate the Hausdorff distances. The total time is the sum of the filter and refine times plus a small amount of additional time required to set up these stages. This table also shows the pruning percentage, which

is the ratio between the number of trajectories that do not belong to the candidate result set and the total database size (60,000 trajectories for GeoLife). The larger the pruning percentage the better.

2.3.5 Parameters

In these experiments, we study the impacts of the parameters: the grid size, the size of the query set ($|P|$), the size of the database ($|Q|$), and K. The range of values and the default value for each of these parameters are presented in Table 2.1.

In each of the following experiments, we study the impact of each of these parameters individually by varying its values in the range of values specified in Table 2.1, while keeping the other parameters fixed at their default values.

Parameter Name	**Range of Values**	**Default Value**
Grid Size	$2 - 256$	64
Query Set Size ($\|P\|$)	$100 - 1000$	500
Database Size ($\|Q\|$)	$10,000 - 60,000$ (GeoLife) $10,000 - 1,700,000$ (Taxi Service)	60,000 (GeoLife) 1,700,000 (Taxi Service)
K	10 - 100	100
Sample Size (TKSimGPU only)	100	100

Table 2.1: Experiment Parameters

The grid size parameter assumes values in a logarithmic scale (e.g., 2, 4, 8, 16). We have chosen 64 as the default grid size because experimentally, this grid size gives consistent, relatively faster results for both algorithms. For the database size (|Q|), we chose 60,000 and 1,700,000 trajectories as the default values for the GeoLife and the Taxi Service datasets, respectively. This decision was made because both techniques, TKSimGPU and FastTopK were designed for dealing with Big Data, so

we are interested in evaluating their performance on the complete datasets.

The sample size is an input parameter that is exclusive to TKSimGPU. This parameter refers to the number of random trajectories that this algorithm picks from the dataset in order to calculate Epsilon. Epsilon is a real number that determines how much to extend the Minimum Bounding Rectangle (MBR) of each trajectory in the query set and is used to prune away all the database trajectories whose MBRs do not intersect or lie in the MBRs of the query trajectories.

Larger sample sizes slow down TKSimGPU because they lead to a larger number of operations in order to compute Epsilon, while smaller sample sizes may render an inaccurate Epsilon. An Epsilon that is either too big or too small has detrimental effects on the query execution time. If it is too big, then TKSimGPU produces larger candidate trajectory sets, which later need to be exhaustive searched; if it is too small, then there is the risk that the candidate result set does not render at least K trajectories for each query trajectory, so the Epsilon value needs to be recalculated. We set the sample size to 100, which was experimentally found to be the best possible value that satisfies the conditions of being fast, because it leads to performing only 100 comparisons, and being accurate enough to not warrant a recalculation of Epsilon.

2.4 Experimental Results

Figure 2.4 shows the overall performance difference between FastTopK and TKSimGPU when their input parameters are set to their default values, presented in Table 2.1. FastTopK produces 3.36X faster execution times on the GeoLife dataset. This is the cumulative performance of running 500 top-K queries on a database of 60,000 trajectories. The reason for this speedup is that FastTopK is able to produce smaller candidate results sets during its filter stage, which results in a faster

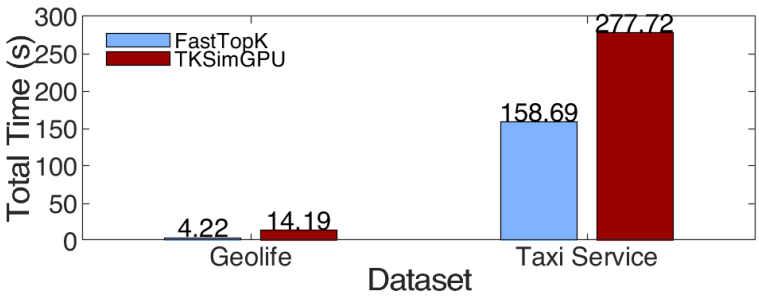

Figure 2.4: Average execution time comparison on both datasets using the default parameters shown in Table 2.1.

calculation of Hausdorff distances.

On the Taxi Service dataset, FastTopK performs 1.75X faster. The reason for this relatively lower speedup is because of the inherent nature of the dataset, where many trajectories span a large number of cells in the grid as taxis go from one part of the city to another. However, FastTopK is still able to calculate the $K = 100$ most similar trajectories from a database of 1.7 million trajectories for 500 queries much faster than TKSimGPU. It lowers the execution time from 277.7 seconds to 158.69 seconds. The speedup and efficiency of the algorithm change for non-default cases, as presented in the next sections, where FastTopK can perform up to 4.7X faster.

In the next section, we discuss the experiments performed to measure the impact of each of these parameters, grid size, K and database size, on the performance of the two algorithms.

2.4.1 Impact of the Grid Size

The grid size is an input parameter shared by both algorithms and specifies the total number of cells that constitute the grid. TKSimGPU uses this parameter to provide an underlying grid on top of which it calculates the increase in the Minimum Bounding Rectangle size by a real number called Epsilon in all directions. These

minimum bounding rectangles are adjusted every time Epsilon is calculated. Next, the intersections between the MBRs of the query and database trajectories are determined and the results are sent to the refine stage.

FastTopK creates a new structure that associates to each grid cell the set of all trajectories that intersect with said cell. The cells intersected by the query trajectory are checked for the database trajectories that lie on these cells, and if the number of resulting trajectories is less than K, then the set of query trajectory cells is expanded to include those grid cells that lie in the periphery of the current set of cells, as explained in Section 2.2.4.

Intuitively we can tell that larger cell sizes would shortlist more trajectories, whereas smaller cell sizes will increase the granularity and produce smaller candidate result sets. This can be seen in the experimental results as well. The smaller the cell size, the smaller the candidate result set, at the expense of greater set-up times.

For this experiment, we used a query set size of 500 trajectories, databases of sizes 60,000 trajectories for GeoLife and 1,700,000 trajectories for Taxi Service, and $K = 100$. Figures 2.5(a) and 2.7(a) show the experiment results for the GeoLife and Taxi Service datasets, respectively. The grid sizes increase over the horizontal axis from 2x2 to 256x256, and the time in seconds is displayed on the vertical axis. To make the axis labels shorter in this figure, the grid sizes are referred to by the number of grid cells along just one of the sides; in other words, a grid size value of 16 in Figure 2.5(a) means that the total number of grid cells is 16x16.

In both datasets, for all grid sizes, FastTopK is considerably faster than TK-SimGPU. For the GeoLife dataset experiment in Figure 2.5(a), FastTopK gives an average speedup of 2.34X, and renders a 3.72X speedup in the best case (grid size 32). For the Taxi Service dataset experiment in Figure 2.7(a), FastTopK provides an average speedup of 1.42X, and renders a 1.75X speedup in the best case. This speedup

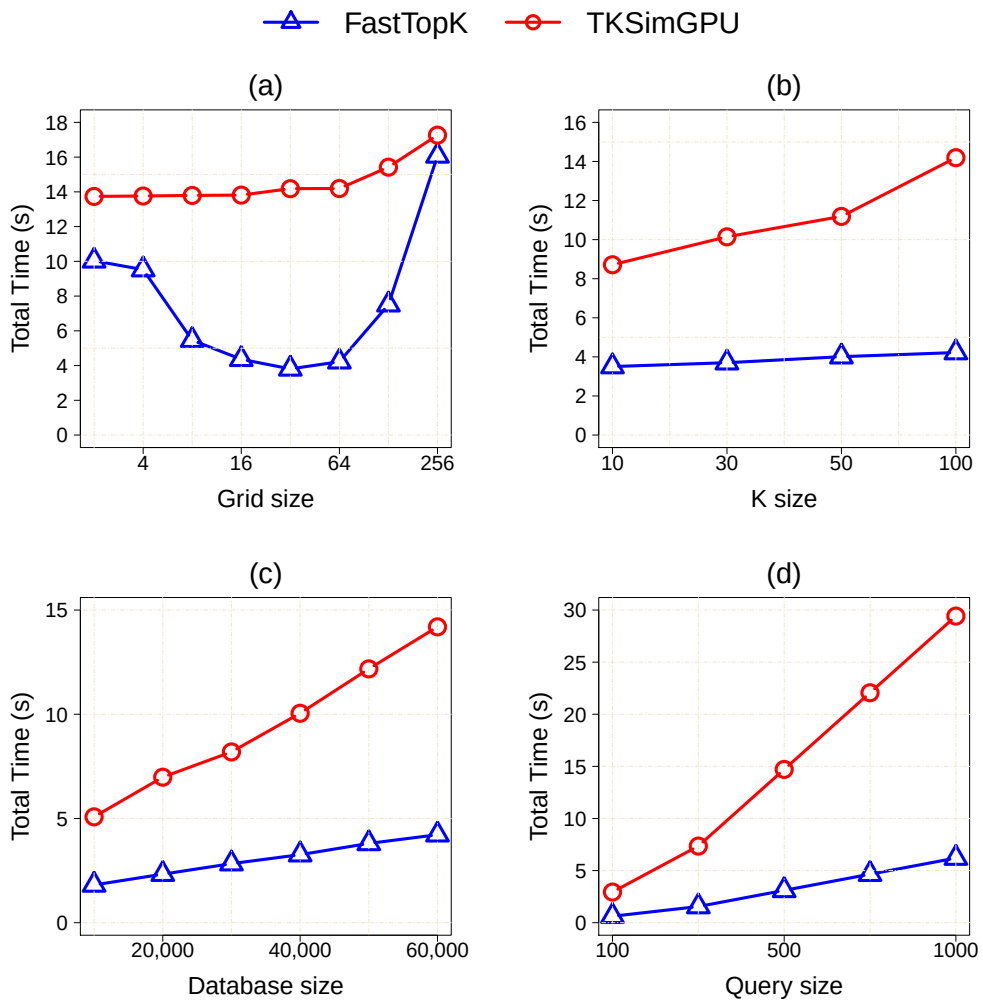

Figure 2.5: Comparison of the execution times of FastTopK and TKSimGPU using the GeoLife dataset. a) Impact of the grid size with query set size = 500, $K = 100$ and database size = 60,000. b) Impact of K with query set size = 500, grid size = 64 and database size = 60,000. c) Impact of the database size with query set size = 500, $K = 100$ and grid size = 64. d) Impact of the query set size with $K = 100$, database size = 60,000 trajectories and grid size = 64.

translates to a difference of 122 seconds between the execution time of TKSimGPU (283 seconds), and that of FastTopK (161 seconds).

We observe that FastTopK has a sweet spot in the range of the measured grid sizes. As seen in Figures 2.5(a) and 2.7(a), the grid sizes of 8, 16, 32 and 64, which

are all in the middle of the tested grid size range, take up less time than the grid sizes on the extreme left and right. This is because large grid sizes correspond to smaller candidate trajectory sets, which makes the refine stage run faster. However, the larger the number of cells, the longer the rasterization process takes to complete.

Grid Cell Size	**Filter time (ms)**	**Refine time (ms)**	**Total time (ms)**	**Pruning %**
TKSimGPU				
2 × 2	4.592	21.923	28.7	0
4 × 4	4.62	21.884	28.71	0.165
8 × 8	4.71	21.91	28.86	0.165
16 × 16	4.77	21.80	28.85	0.73
32 × 32	5.55	21.75	29.56	0.96
64 × 64	5.57	21.77	29.63	1.036
128 × 128	8.01	21.80	32.13	1.063
256 × 256	11.66	21.82	35.83	1.085
FastTopK				
2 × 2	4.044	14.434	18.48	20.86
4 × 4	4.179	13.516	17.69	26.26
8 × 8	4.58	2.86	7.45	88.66
16 × 16	4.731	1.107	5.96	96.93
32 × 32	4.826	0.885	5.61	98.01
64 × 64	6.051	0.668	7.551	99.08
128 × 128	11.178	0.626	14.297	99.34
256 × 256	29.61	0.599	32.57	99.44

Table 2.2: Impact of the Grid Size on the Execution Time over the GeoLife dataset with 60,000 Trajectories ($K = 100$, $|P| = 1$)

Table 2.2 shows, for both FastTopK and TKSimGPU, the filter time, refine time, total execution time and pruning percentage, which is the percentage of trajectories pruned from the initial dataset. The results in Table 2.2 correspond to processing a single query trajectory. The reason for the speedup of FastTopK, and the cause of the sweet spot are both apparent from this table. The filter stage of FastTopK is much more effective than TKSimGPU's, as evident from the pruning power of both.

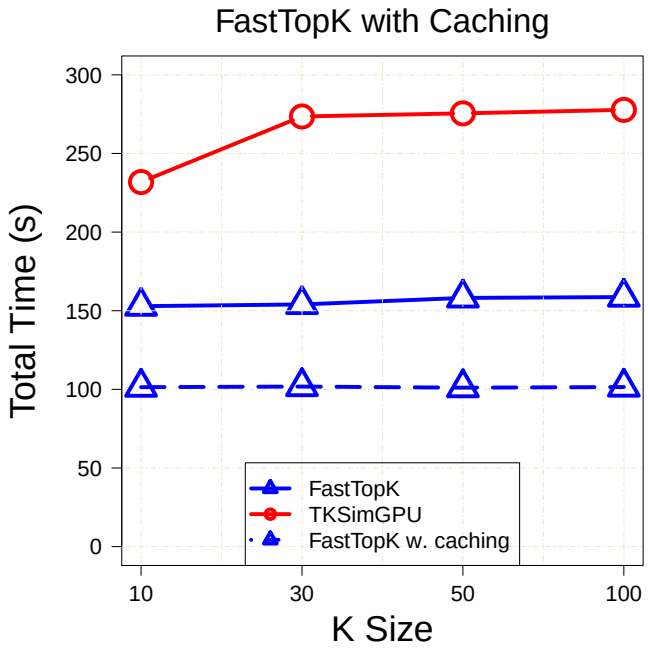

Figure 2.6: Impact of K over execution time on Porto dataset with database rasterization done once.

Moreover, for smaller grid sizes, the classification time and filter time are also lesser than those of TKSimGPU's; but with the increase in grid size, the classification of FastTopK increasingly takes more time. Even though a bigger grid size leads to a shorter refine time, by making the pruning process more effective, after a certain grid size, the classification overhead of the larger number of cells dominates the total execution time. This is the reason behind the sweet spot's existence. However, despite this tradeoff, FastTopK shows a shorter total execution time than TKSimGPU for any grid size in our experiments.

2.4.2 Impact of K

For this experiment, we used a query set of 500, databases of sizes 60,000 and 1,700,000 for the GeoLife and Taxi Service datasets, respectively, and a grid size of

64x64.

Figures 2.5(b) and 2.7(b) show the results of this experiment. In these figures we observe that both TKSimGPU and FastTopK increase their execution times as K increases.

However, as is evident from these figures, the rate of increase of TKSimGPU's execution time is greater than that of FastTopK. The average speedup obtained by FastTopK over TKSimGPU is 2.84X for the GeoLife dataset, and 1.69X for the Taxi Service dataset.

As seen in the Algorithm 2.1, the while loop in Line 12 does not conclude until the number of trajectories in the query result set is equal to K. This means that increasing K could potentially increment the number iterations in this while loop. Moreover, a bigger K has a higher tendency to make the candidate result sets bigger, hence resulting in longer refine times.

In TKSimGPU, a larger K results in a larger Epsilon value, which creates bigger MBRs. A bigger MBR brings in more trajectories for refining, which leads to longer execution times. It is evident from the results that FastTopK does a more efficient pruning job, even with larger values of K.

It is interesting to note that the rasterization of the database needs to be done only once. This means that, if the grid size and the database are fixed, then running FastTopK for different values of K can achieve even higher speedups over TKSimGPU. This is not possible in TKSimGPU, since it re-computes the grid every time any of the input parameters changes. Figure 2.6 demonstrates the results of the experiment on the Taxi Service dataset if we utilize the already rasterized database and run only the refine stage for every K size. As is evident from this figure, the speedup of FastTopK over TKSimGPU increases from 1.69X to 2.61X if this feature is turned on.

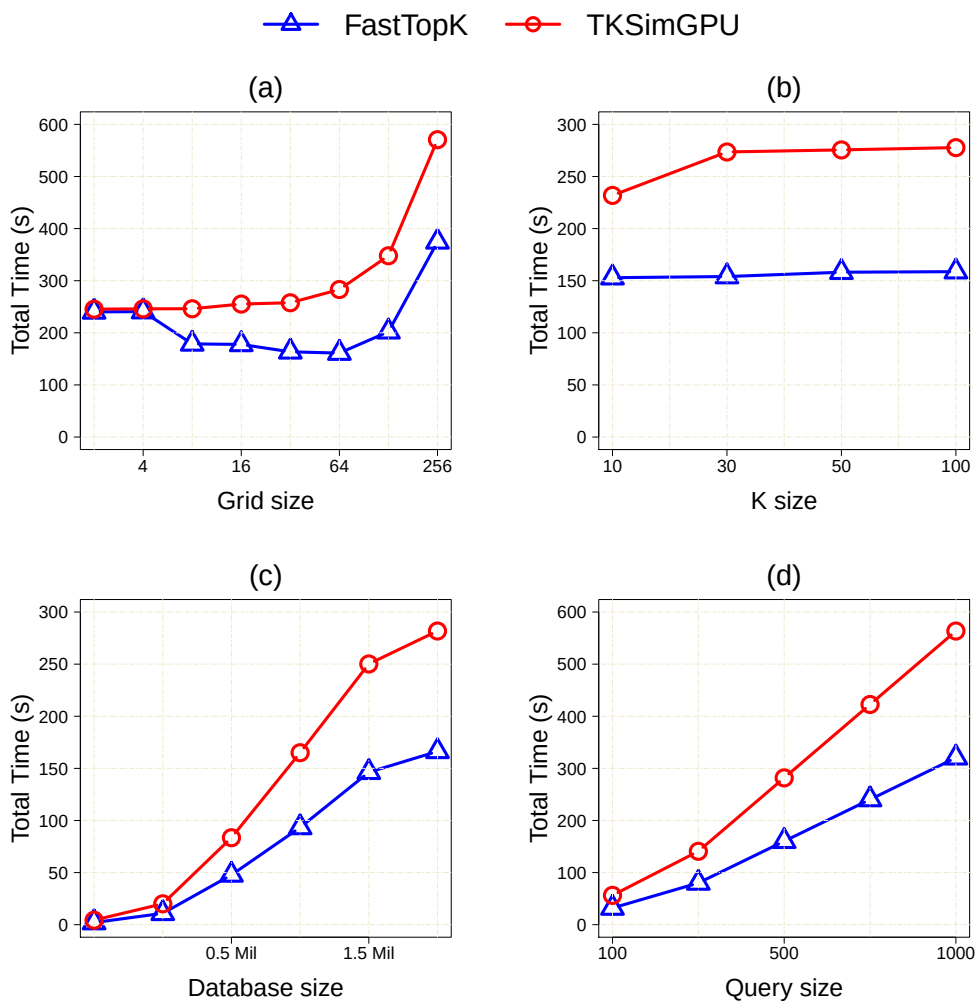

Figure 2.7: Comparison of the execution times of FastTopK and TKSimGPU using the Taxi Service Trajectory dataset. a) Impact of the grid size with query set size = 500, $K = 100$ and database size = 1,700,000. b) Impact of K with query set size = 500, grid size = 64 and database size = 1,700,000. c) Impact of the database size with query set size = 500, $K = 100$ and grid size = 64. d) Impact of the query set size with K =100, database size = 1,700,000 and grid size = 64.

2.4.3 Impact of the Size of the Database

For this experiment, we fixed the value of K to 100, used a query set of size 500, and a grid size of 64×64.

Figures 2.5(c) and 2.7(c) show the results of this experiment. Both TKSimGPU

and FastTopK show an increase in execution time as the database size grows, but FastTopK provides consistent and increasing speedup for TKSimGPU with the increase in database size. Overall, the speedup achieved is on average 3.05X on GeoLife, and 1.82X on the Taxi Service dataset. Because of the inherent structure of TKSimGPU, the Epsilon returns a greater number of trajectories when the database size is larger. This is not a problem in FastTopK, because it does not rely on Epsilon. Not only does it classify the trajectories efficiently, it also does more effective filtering, which in turn outputs smaller candidate trajectory sets. These smaller candidate trajectory sets result in faster execution times for the refine stage.

2.4.4 Impact of the Size of the Query Set

This experiment exhibits the scalability of the FastTopK algorithm compared with TKSimGPU's when the size of the query set increases. In this experiment, we executed both algorithms with query set sizes ranging from 100 queries to 1,000 queries, with K set to 100, a grid size fixed at 64, and database sizes of 60,000 and 1,700,000 for the GeoLife and the Taxi Service datasets, respectively.

Figures 2.5(d) and 2.7(d) show the results of this experiment. FastTopK showcases higher speedups than TKSimGPU when the query set size grows. For a query set size of 1,000, TKSimGPU completes its execution in 29.4 seconds, whereas FastTopK takes only 6.2 seconds for the GeoLife dataset; this is a 4.7X speedup. With the Taxi Service dataset, the difference is even greater with the execution time being 563.6 seconds and 320.4 seconds for TKSimGPU and FastTopK, respectively. This speedup can be attributed to the shorter execution time of a single query, which is in turn due to shorter filter and refine times. Intuitively, it can be expected that for even larger sizes of the database, the query set, or for larger values K, the performance gap

between FastTopK and TKSimGPU will widen; this is because the rate of increase of the execution time of FastTopK with respect to the number of queries is far lower than that of TKSimGPU's.

2.4.5 Memory Impact

Figure 2.8 shows the total memory consumed by both algorithms. Nvidia GTX 1060, the GPU used for this experiment, shows a maximum memory capacity of 6,069 MB. This is obtained, along with the following information, by using CUDA's `cudaMemGetInfo` API that returns the total available memory and free device memory available.

After transferring the dataset to the GPU, around 709 MB of GPU memory is consumed. This includes all the longitude and latitude points in a contiguous order with another array marking the beginning of the next trajectory. The processing of this data is done in multiple steps, and each of these steps allocate and deallocate the memory. For example, Hausdorff distance calculation and sorting. But the step that requires the highest amount of memory and thus provides us with an upper bound on the memory needed to run the algorithm is the rasterization step in line 16 in algorithm 2.1. This is where for every trajectory, the number of cells it occupies is returned. It takes up the greatest amount of memory of any step because it takes place before the pruning, so the GPU deals with the entire dataset.

As expected, the amount of memory consumed scales with the number of cells (grid size) and the number of trajectories (database size). Figures 2.8 (a) and (b) show the effect of both these parameters respectively for the Taxi service dataset. In all the cases, FastTopK consumes slightly more memory than TKSimGPU. This is because the output of the rasterization function on TKSimGPU returns fewer cells

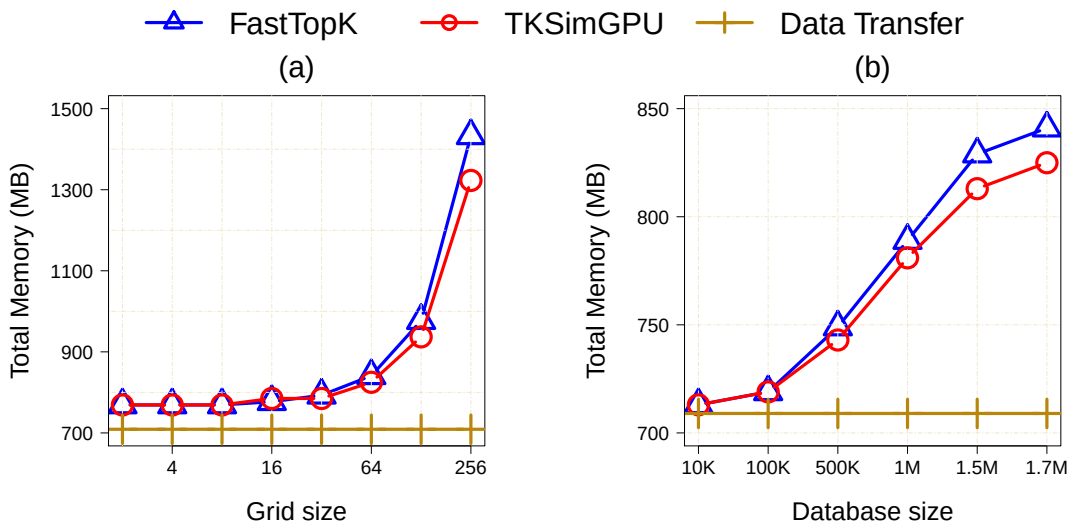

Figure 2.8: Comparison of the total memory consumption of FastTopK and TKSimGPU using the Porto Taxi Dataset (a) Impact of grid size on memory. (b) Impact of database size on memory.

than FastTopK does. This takes place because the TKSimGPU algorithm scales the global spatial area the grid is applied on by a variable *Epsilon*, which is the radius of the K closest trajectories it uses for pruning. This increase in area makes the trajectories fall into fewer cells, and this effect increases with the increase in Epsilon. The less tight rasterization of the trajectories may be a cause for poorer pruning power of TKSimGPU, but it does save the algorithm a small amount of extra memory that FastTopK requires.

2.5 Discussion

To summarize, from the experiments described in this chapter we can conclude the following:

- Using a novel, grid-based pruning strategy, we developed a new technique, FastTopK [46], that safely prunes trajectories that are sure to be not a part of the

resulting dataset.

- The candidate sets produced by our technique were 176.63X smaller than the state of the art pruning technique, TKSimGPU [35]. The result was a considerably smaller dataset (up to 99.44% smaller) that had fewer trajectories that the GPU could exhaustively search in parallel.

- Upon evaluating the impact of the parameters required for the execution of FastTopK and TKSimGPU, we observed that the execution time increased with the increase in grid size, K, database size, and query set size in both the techniques. However, FastTopK processed the queries faster than TKSimGPU over all the observed values of these parameters, going as fast as 4.7X in the best case.

- The memory footprint of both the techniques showed that FastTopK did require more memory to operate than TKSimGPU. However, the difference was marginal, going up to 1.08X higher in its worst case.

- Thus, with cheap and efficient pruning, FastTopK, a top-K trajectory search algorithm, experimentally proven to be up to 4.7X faster than the next fastest algorithm, TKSimGPU [35], which itself is 3.8X faster than a multicore CPU implementation.

3 DBSCAN Clustering: Contemporary GPU-based Algorithm Analysis

In this chapter, we present the first experimental evaluation of GPU-based DBSCAN algorithms. We evaluate these on three real-world datasets on the same hardware and experimentally determine the memory and processing time impact of parameters like *Epsilon*, *minPts* and dataset size. Additionally we discuss the inherent features of these techniques that results in their particular performance. Section 3.1 presents related work. Section 3.2 describes all the techniques we tested in detail and Section 3.3 and 3.4 provide an experimental evaluation of these techniques.

3.1 Related Work

3.1.1 Clustering

The existing algorithms for cluster analysis can be broadly categorized into the following groups: partitioning methods, hierarchical methods, density-based methods, and grid-based methods [22]. *Partitioning methods* find clusters by iterative relocation, which consists in that points are initially assigned to tentative clusters, and then they are iteratively moved to different clusters until settling for a good as-

signment of points to clusters. Examples of partitioning methods are K-means [41] and K-medoids [29]. *Hierarchical methods* do not work by iterative relocation, but instead produce a hierarchical decomposition of the data called a taxonomy. Some hierarchical methods, called agglomerative methods, work by considering each point as its own cluster, and then iteratively merging the two most similar clusters until there is one cluster. Examples of hierarchical methods are hierarchical agglomerative clustering and hierarchical divisive clustering [22]. *Density-based methods* do not work by iterative relocation or hierarchical decomposition, but instead, define clusters to be areas of high density, which allows these methods to find non-globular clusters. Examples of density-based methods are DBSCAN [13] and OPTICS [4]. *Grid-based methods* partition the space by using a grid and the clustering operations are performed on the cells of this grid. Examples of grid-based methods are STING [60] and CLIQUE [2].

3.1.2 DBSCAN

For a time it was believed that the worst-case time complexity of DBSCAN could be reduced to $O(n \cdot \log n)$ by using indexes like R-trees [20], but this was shown not to be the case in the work of Gan and Tao [16], whose result has encouraged several efforts to improve the execution time performance of DBSCAN. Some of these works have proposed parallel algorithms for multicore CPUs, like the works of Götz, Bodenstein and Riedel [18] and Patwary et al. [51], and some have proposed Map-Reduce approaches like Dai and Lin [10]. Other works, instead of proposing new parallel algorithms for DBSCAN clustering, have focused on designing new single-threaded DBSCAN algorithms that improve upon DBSCAN's quadratic worst-case time complexity, like DBSCAN++ [28].

3.2 Competing Techniques

In this section we describe the DBSCAN algorithms studied in this paper: a single-threaded CPU DBSCAN algorithm (DBSCAN) [13]; a multi-threaded CPU DBSCAN algorithm, and three GPU DBSCAN algorithms, Thapa et al.'s algorithm [59], CUDA D-Clust [6], and G-DBSCAN [3].

3.2.1 DBSCAN

DBSCAN by Ester et al. [13] is a density-based clustering algorithm that requires, besides the dataset D that is going to be clustered, two input parameters, *eps* (also called *epsilon*) and *minPts*. DBSCAN makes use of the concept of *epsilon-neighborhood* of a point to cluster a dataset. The *epsilon-neighborhood* of a point p consists of all the points in the dataset D (including p itself) whose Euclidean distances from p are less than or equal to *eps*. All the points in D such that their epsilon-neighborhoods contain at least *minPts* many points belonging to D are labelled as *core points*; all the points in D that are not core points, but belong to the epsilon-neighborhood of a core point are labelled as *border* points; and all other points in D that are neither core nor border points are labelled as *noise* points. After labelling each point in D, DBSCAN proceeds to find clusters.

Before describing how DBSCAN finds clusters, we explain two mathematical relations on which DBSCAN relies for this purpose: direct density reachability and density connectedness. Given two points p and q in the dataset D, q is said to be *direct density reachable* from p, denoted by $p \triangleleft q$, if p is a core point and q is in the epsilon-neighborhood of p.

Given two points p and q in D, they are said to be *density connected*, denoted by $p \bowtie q$, if there exists a sequence of m core points $p_1, p_2, \ldots, p_m$ in D such that

p is direct density reachable from p_1, and q is direct density reachable from p_m and $p_1 \triangleleft p_2 \triangleleft ... \triangleright p_m$.

A *cluster*, according to DBSCAN, is a maximal set C of points contained in D such that for every pair of points p and q contained in C, they are density connected. In this context, *maximal* means that there does not exist a different subset E of points in D such that E contains C, and every pair of points in E is density connected.

Algorithm 3.1: DBSCAN Algorithm

Input: Dataset D, *eps*, *minPts*
Output: Each point p in D has label(p) indicating its cluster ID

```
1  procedure DBSCAN(D, eps, minPts):
2      clusterID ← 0;
3      for each point p in D do label(p) ← unclassified;
4      for each point p in D do
5          if label(p) = unclassified then
6              if ExpandCluster(D,p,eps,minPts,clusterID) then
7                  clusterID ← clusterID + 1;

8  function ExpandCluster(D,p,eps,minPts,clusterID)
9      neighbors ← RangeQuery(D,p,eps);
10     if |neighbors| < minPts then
11         label(p) ← noise;
12         return false;
13     else                                          // p is a core point
14         label(p) ← clusterID;
15         stack ← neighbors − {p}; // insert p's neighbors into stack
16         while stack is not empty do
17             top ←remove top elements from stack ;
18             if label(top) = noise then label(top) ← clusterID;
19             if label(top) ≠ unclassified then continue;
20             label(top) ← clusterID;
21             newNeighbors ← RangeQuery(D,top,eps);
22             if |newNeighbors| ≥ minPts then
23                 insert newNeighbors into stack;

24 DBSCAN(D,eps,minPts);
```

We can express the above definition of border point in terms of these two mathematical relations. A border point is a point that is not a core point, but that is density-connected to some core point. The density-connectedness relation is symmetric, meaning that if p is density connected with q, then q is also density connected with p. Moreover, a non-core point p can be density connected to a non-core point q if they both lie in the epsilon-neighborhood of a core point r.

Since several of the GPU algorithms studied in this chapter have a structure that is very similar to that of DBSCAN, we present the pseudo-code of DBSCAN in Algorithm 3.1. In Lines 1 to 3, the algorithm performs its initialization steps, where all the points in D are labelled as "unclassified." Then in Lines 4 to 7, the algorithm selects an unclassified point p and calls the `ExpandCluster` function using p as one of its input parameters. The purpose of the `ExpandCluster` function is determine the set of all points in D that are density connected to p. This function determines if p is a core point (Lines 9 to 13), and if it is, the function will change the label of p to denote that p is a core point (Line 14), and then add all the points in p's epsilon-neighborhood to a stack (Line 15). Lines 16 to 23 will iteratively select the top element of this stack, then add all the points in the epsilon-neighborhood of this point to this stack; the algorithm proceeds in this manner until the stack is empty.

3.2.2 Thapa et al.'s Algorithm

Thapa, Trefftz and Wolffe's [59] is a GPU algorithm for DBSCAN clustering. Its key idea consists in leveraging the high instruction throughput of GPUs to perform range queries for every point in the dataset. This algorithm is very similar to DBSCAN [13] (See Algorithm 3.1), with the difference that the `RangeQuery` function calls (in Lines 9 and 21 of Algorithm 3.1) are replaced by calls to the

`MultiRangeQuery` function shown in Algorithm 3.2. In this algorithm we see that the function `MultiRangeQuery` initializes the result set with the empty set (Line 2), then it uses multiple GPU threads to find the epsilon-neighborhood of *point* (Lines 3 to 6). The epsilon-neighborhood of *point* is computed by assigning the points in the dataset D to different GPU threads, which then check in parallel if these points are at a distance less than or equal to *eps* from *point*. Since Thapa et al.'s algorithm does not use R-trees [20] for computing the epsilon-neighborhood of each point, but instead, chooses to perform exhaustive range queries over the dataset D, we see that the average and worst-case amount of work that each *range query* performs is $O(n)$, where n is the size of the dataset D.

Thapa et al.'s algorithm is a slight modification of a similar algorithm developed by Trefftz et al. [59]. This algorithm of Trefftz et al. instead computes the matrix M containing the pairwise distances between every point in the dataset; this matrix satisfies that for any two points i and j belonging to the dataset D, M_{ij} is the distance between i and j. This matrix is computed by the Trefftz et al. algorithm in parallel by assigning each entry in M to a different GPU thread, and this matrix requires $O(n^2)$ amount of GPU RAM.

Algorithm 3.2: Thapa et al. and multi-threaded CPU Algorithm

```
1 function MultiRangeQuery(D,point,eps)
2     result ←empty set;
3     for each point p in D do in parallel
4         dist ← distance between point and p;
5         if dist ≤ eps then
6             add p to result;
7     return result;
```

An advantage of Thapa et al.'s algorithm over Trefftz et al.'s algorithm is that the former requires $O(n)$ amount of RAM on the GPU, while the latter requires $O(n^2)$.

However, Thapa et al.'s algorithm has the disadvantage that it performs more kernel calls than Trefftz et al.'s algorithm. This is because the former algorithm performs one kernel call for every range query, while the latter makes a single kernel call during the execution of the whole algorithm. This greater number of kernel calls also implies that Thapa et al.'s algorithm makes more memory transfers to and from the GPU, which carry additional performance penalties. Since in the experiments presented in Thapa et al. [59], it was shown that the Thapa et al.'s algorithm was superior in both execution time and memory requirements, we do not include the Trefftz et al.'s algorithm in our comparison.

3.2.3 Multi-threaded CPU DBSCAN

Multi-threaded CPU DBSCAN, which we also call multi-threaded CPU in this chapter, is a parallel algorithm for DBSCAN clustering designed to run on multicore CPUs. This algorithm has the same structure and parallelization strategy of Thapa et al.'s algorithm, in which each of the `RangeQuery` function calls (in Lines 9 and 21 of Algorithm 3.1) of DBSCAN is replaced by calls to the `MultiRangeQuery` function shown in Algorithm 3.2 which make use of different threads to compute the *epsilon-neighborhood* of a point. The only difference with Thapa et al.'s algorithm is that it uses CPU threads instead of GPU threads. This algorithm was introduced in this paper with the purpose of comparing it against Thapa et al.'s algorithm.

3.2.4 CUDA-DClust

Böhm et al. proposed CUDA-DClust [6], which is a GPU algorithm for DBSCAN clustering based on the concept of chains. A *chain* is a subset of a DBSCAN cluster, and as such, a cluster can have multiple chains, but a chain belongs to only one

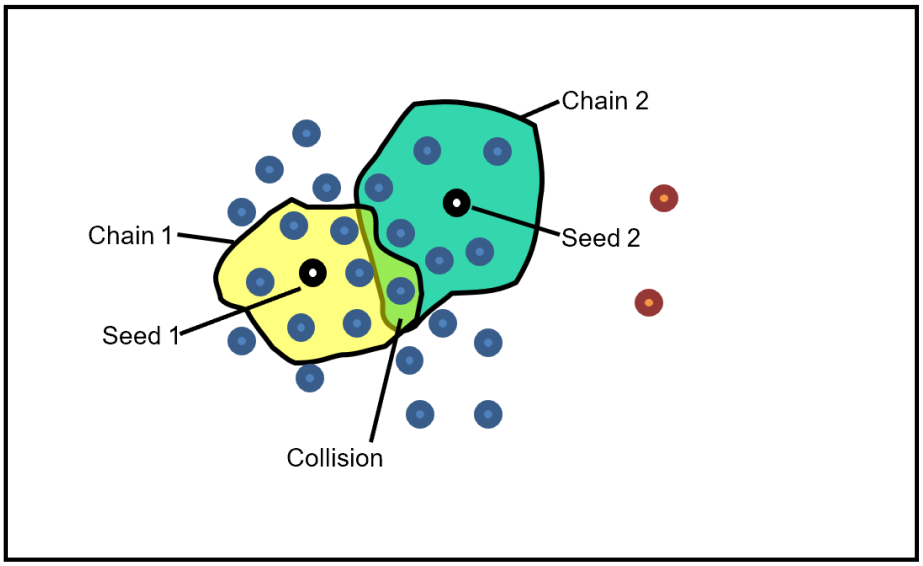

Figure 3.1: Two chains starting off from different seed points in the same dataset can collide and become one cluster.

cluster; in other words, a chain C is a subset of points of the dataset D such that all points in C are density connected to each other, but C is not necessarily maximal. Hence, each chain is essentially a subset of a cluster that will incrementally grow, by adding more points to itself, to be a complete cluster. In the beginning stages of CUDA-DClust, each chain consists of a single point and is assigned to a different GPU block, and then all the chains are expanded in parallel.

When the chains grow in parallel, there is the possibility that two chains collide. In this case, the colliding chains need to be merged into a single cluster. To deal with this, CUDA-DClust uses a dedicated *collision matrix* that each chain updates if it finds a point to be already labeled as belonging to a different chain. This assignment and checking use atomic operations since multiple threads may try to label different chains.

Figure 3.1 shows the process of parallel expansion of chains. Two seed points, Seed 1 and Seed 2 are selected from the dataset on which CUDA-DClust algorithm

runs in parallel, adding their neighbors recursively to a chain. Seed 1 expands Chain 1 whereas Seed 2 expands Chain 2 in Figure 3.1. These chains may collide, as shown in Figure 3.1, due to a chain extending to a point already classified as member of a different chain. When this occurs the *collision matrix* classifies these colliding chains as member of the same cluster.

The pseudo-code of CUDA-DClust is shown in Algorithm 3.3. This algorithm begins by launching (Line 33) the `ClusterExpansion` kernel (Line 1). Each GPU block allocates space for its own *neighborCount* integer and its own *quarantine* array, which are both shared across the threads within each block. Each GPU block (note that each GPU block is associated with exactly one chain and vice versa) maintains an array called *seedList* (Line 2), which stores the neighbors of a core point, and whose maximum size is *maxSeedLength*. Each GPU block is also in charge of reading a single point p from the dataset, so that every point in the dataset D is read by exactly one GPU block (Line 5). After this, a barrier synchronization instruction (Line 6) ensures that inside every GPU block, no thread proceeds until all threads in that block have read the point p from the GPU's global memory. Each thread in every GPU block is in charge of calling `ProcessObject` with a different point q in the dataset (Lines 7 to 8).

The `ProcessObject` procedure (Line 16) computes the distance between q and p (note that p is stored in the GPU's shared memory), and if q is in the epsilon-neighborhood of p (Lines 17 and 18), then it performs an atomic increment (Line 19) on the variable *neighborCount*, which is a variable in shared memory used to determine the size of the epsilon-neighborhood of p. If p is found to be a core point (Line 20), then q is indeed part of the same chain as p, so the algorithm calls the `MarkAsCandidate` function on q; otherwise, p might not be a core point, so it is placed in quarantine (Line 21).

Algorithm 3.3: CUDA-DClust Algorithm

Input: Dataset D, eps, $minPts$

Output: Each point p in D has label(p) indicating its cluster ID

```
1  kernel ClusterExpansion(D,eps,minPts)
2      seedList[0 .. maxLen]; // array shared by all threads in a block
3      neighborCount ← 0;
                          // variable shared by all threads in a block
4      quarantine[0 .. minPts − 1];  // array shared by all threads in a
        block
5      Each GPU block reads from global memory in a different point p from D.
        All the threads in each block share their block's p.
6      Barrier synchronization in each block;
7      for each point q in dataset D do in parallel
8          ProcessObject(q);
9      Barrier synchronization in each block;
10     if neighborCount ≥ minPts then
11         label(p) ← clusterID;
12         for i in the interval [0, minPts) do
13             MarkAsCandidate(quarantine[i])
14     else
15         label(p) ← noise;

16 procedure ProcessObject(q)                    // p is in shared memory
17     dist ← distance between p and q;
18     if dist ≥ eps then
19         h ← atomic increment on neighborCount ;
20         if h ≥ minPts then MarkAsCandidate(q);
21         else quarantine[h] ← q;
```

```
22 procedure MarkAsCandidate(q)                    // p is in shared memory
       // CAS: If label(p) is unclassified, then atomically replace
      it with clusterID.
23     oldLabel ← CAS(label(p), unclassified, blockID);
24     if oldLabel = unclassified then
25         h ←atomic increment of mySeedLength ;
26         if h < maxLen then
27             seedList[h] ← q;
28     else                                        // p has been classified before
29         if oldLabel ≠ noise and oldLabel ≠ blockID then
30             collisionMatrix[oldLabel][q] ← true;
31         else
32             collisionMatrix[q][oldLabel] ← true;

33 ClusterExpansion(D,eps,minPts);
```

The `MarkAsCandidate` procedure (Line 22) receives as an input parameter the point q, which is in the epsilon-neighborhood of the point p. Note that p has just been shown to be a core point. Then, CUDA-DClust checks if p has been labelled before, and if it has not, then it is labelled as core by assigning to it a *clusterId*, which is the identifier of the GPU block (Line 23). If p was an unclassified point until the previous instruction, it is the first time that p has been discovered as a core point (Line 23), then we atomically increment the number of neighbors of p (Line 24) and add q as a neighbor to p (Line 27).

If p has been classified before (Line 28) and it belongs to the same chain that the current thread block is expanding, then nothing else needs to be done. Otherwise, if p has been classified before and if it was shown to belong to another chain (Line 29 to 32), then there is a collision between the two chains; therefore, we need to mark that the two chains that collided form part of the same cluster.

With this algorithm it may be the case that some chains finish processing before others, which means that some GPU blocks might remain idle as soon as they process

their corresponding chains. To deal with this possible load imbalance, the algorithm forces that idle block to start expanding a new chain from the available unused seeds. If it does not find any, it splits the existing chain expansions between two or more GPU blocks.

A variation of CUDA-DClust, called CUDA-DClust* [6], employs an indexing structure to improve the execution time. This index structure uses a constant number of directory level partitions. At each level, the data is sorted according to a specific dimension (the first dimension at first level, the second dimension at the second level, and so on). After sorting at each level, the data is partitioned into a pre-determined number of constant bins after which within that partition, sorting takes place according to the next dimension. Experimental evaluation [6] has shown this indexing to speed up the algorithm, thus in this work we have used the faster, indexed version for our experiments, and for simplicity we call it CUDA-DClust instead of CUDA-DClust* for the remainder of this paper.

3.2.5 G-DBSCAN

Andrade et al. proposed G-DBSCAN [3] for DBSCAN clustering on GPUs. The key idea behind this algorithm consists in using GPUs to build the density connectedness graph corresponding to the dataset D, and then performing parallel multiple BFS searches over this graph.

Given a dataset D, its *density connectedness graph* is an undirected graph *(V, E)* whose set of nodes or vertices V contains every point in the dataset D, i.e., there is exactly one node in V for every point in D. The set of edges E of the graph is defined as follows. If a point (node) p is within a distance of *eps* from another point (node) q, then there exists an edge e in E that connects the node p with the node q. Since

the set of nodes of the density connectedness graph is the same as the set of points in the dataset D, then in the following discussion of G-DBSCAN we will use the terms *node* and *point* interchangeably.

From the definition of density connectedness graph, it can be seen that each DBSCAN cluster corresponds to exactly one connected component in this graph and vice versa. Therefore, to find the DBSCAN clusters of a dataset D, it is enough to find the connected components of the density connectedness graph of D, which is precisely what G-DBSCAN does by performing successive breadth-first searches (BFS) on the graph [23].

The pseudo-code of G-DBSCAN is shown in Algorithm 3.4. There we see that G-DBSCAN receives the dataset D, *minPts*, and *eps* as input parameters. Then, G-DBSCAN proceeds to build the density-connectedness graph by calling the function `MakeGraph` (Line 1 of Algorithm 3.4) in Line 33. To reduce the amount of global memory required on the GPU and help ensure memory coalescing, a compact adjacency list representation is used. This representation consists of two arrays, *adjacency* and *startPos*, that satisfy that for any point p_i in the dataset D, the nodes that are adjacent to p_i in this density-connectedness graph will be stored consecutively in *adjacency[startPos(p_i) ... startPos(p_{i+1}) - 1]*, so that *startPos(p_i)* marks the position in the adjacency array for the first node that is adjacent to p_i.

Lines 2 to 4 use GPU threads in parallel to compute the size of the epsilon-neighborhood of every point p in the dataset D, and then these sizes are stored in the *numNeighbors* array, so that *numNeighbors[i]* contains the number of points in the epsilon-neighborhood of point i. In Line 5, a parallel exclusive prefix sum is performed over the *numNeighbors* array and then stored in the *startPos* array. Lines 6 to 8 find the edges of the density connectedness graph by checking if every pair of points p and q in D is density connected (Line 7), and if they are, then q is added to the adjacency

list of p and vice versa. Then, Line 9 returns the density-connectedness graph.

After the density connectedness graph has been created in the GPU, G-DBSCAN calls (Line 34) the `IdentifyClusters` function (Line 26). This function will mark every node v in the graph as *not visited* (Line 28), and then for every core point that has not been visited, it will call the `GPU-BFS` function (Lines 29 to 32) using that point as a starting point. Every call to the `GPU-BFS` function with a node v as its input parameter finds the whole cluster associated with node v.

Algorithm 3.4: G-DBSCAN Algorithm

Input: Dataset D, eps, $minPts$
Output: Each point p in D has label(p) indicating its cluster ID

```
1  function MakeGraph(D,p,eps,minPts,clusterID)
2      for each p in D do in parallel numNeighbors[p] ← 0
3      for each pair(p,q) of points in D do in parallel
4          if dist(p,q) ≤ eps then numNeighbors[p] + +;
5      startPos ←ExclusivePrefixSum(numNeighbors);
6      for each pair(p,q) of points in D do in parallel
7          if dist(p,q) ≤ eps then
8              Add q to the adjacency list of p and vice versa;
9      return new Graph(adjacency,startPos);


10 kernel GPU-BFS-Kernel(graph,eps,minPts,frontier,visited)
               // Each thread, threadId, checks if its node is in the
       frontier.
11     if frontier[threadID] then
12         frontier[threadID] ←false; visited[threadID] ←true;
13         for each neighbor n of the node with identifier threadID do
14             if not visited[n] then frontier[n] ←true

                                          // A procedure that runs on the GPU
```

```
15 procedure GPU-BFS(v,graph,eps,minPts,clusterID)
16     Initialize array F[1..graph.numNodes] with false values;
17     Initialize array V[1..graph.numNodes] with false values;
18     F[v] ←true;// Put node v in the frontier
19     while F has some node with a value of true do
20         GPU-BFS-Kernel(graph,eps,minPts,F,V);
21     Bring the V array from the GPU to the host;
22     for each node n in the graph do
23         if V[n] then
24             label(n) ← clusterID;
25             visited(n) ←true;

26 procedure IdentifyClusters(graph,eps,minPts)
27     clusterID ← 0;
28     for each node v do visited(v) ← false
29     for each node v in the graph do
30         if not visited(v) and v is a core point then
31             visited(v) ←true; label(v) ← clusterID;
32             GPU-BFS(v,graph,eps,minPts,clusterID++)

33 graph ← MakeGraph(D,eps,minPts)
34 IdentifyClusters(graph,eps,minPts)
```

The GPU-BFS function (in Line 15) is in charge of running the BFS algorithm starting from a node v. This function initializes two arrays F and V in the GPU's global memory with *false* values (Lines 16 and 17). The array F represents the *frontier* of BFS, i.e., those nodes that are immediately adjacent to visited nodes. More concretely, if *F[n]* is true for some node n, then that means that node n belongs to the frontier. The array V represents the nodes that have been visited by the current call to GPU-BFS, so that if *V[n]* is true for some node n, then that means that node n has been visited by the current call to GPU-BFS. Then the node v is placed in the frontier (Line 18) and as long as there are nodes in the frontier, the

`GPU-BFS-Kernel` function will be iteratively called (Lines 19 to 20). This loop will finish when there are no more nodes in the frontier F, and this happens exactly when the whole cluster corresponding to v has been explored by BFS. After this, the V array, containing true values for all those nodes that are density reachable from v, is sent from the GPU to the host (Line 21), and all those nodes that are reachable from v are labelled as *visited* and as belonging to the cluster of v (Lines 22 to 25).

The `GPU-BFS-Kernel` is a kernel function, meaning that all its instructions are executed in parallel by the GPU threads. To simplify the discussion of this algorithm, we assume that the number of GPU threads is equal to the number of nodes in the graph; however, it is easy to modify the pseudocode to take this into account. If this is the case, then the GPU threads can be assigned identifiers from 0 to $n-1$, where n is the number of points in the dataset D; similarly, the points in D can also be assigned identifiers from 0 to $n-1$. As a consequence of this assignment of identifiers, each GPU thread with identifier i is associated with the node with identifier i. Then, each thread with identifier *threadId* checks if its associated node belongs to the frontier (Line 11), and if it does, it will mark it as *visited* and remove it from the frontier (Line 12). After this, each thread *threadId* will find all the nodes that are adjacent to its associated node, and if these adjacent nodes have not been visited yet, they are added to the frontier (Lines 13 to 14).

3.3 Performance Analysis

In this section, we describe the experiments conducted to evaluate the performance of the existing GPU-based algorithms for DBSCAN clustering.

3.3.1 Datasets

For our experiments we used three real world datasets. All these datasets are spatial and two-dimensional. We now describe each of these datasets.

The first one is the Taxi Service Trajectory Prediction Challenge dataset [44]. It consists of the trajectories of 442 taxis running in the city of Porto, Portugal, over the period of one year. This is a dataset with over 1,710,000+ trajectories with 81,000,000+ points in total. Every data point in this dataset has, besides longitude and latitude values, a unique identifier for each taxi trip, taxi ID, timestamp, and user information.

The second dataset we used is the 3D Road Network dataset [30]. It consists of more than 400,000 points from the road network of North Jutland in Denmark. Each data point contains its ID, longitude, latitude, and altitude.

The third dataset used is the Next Generation Simulation (NGSIM) Vehicle Trajectories dataset [58]. It consists of vehicle trajectory data collected by NGSIM researchers on three highways in Los Angeles, CA, Emeryville, CA, and Atlanta, GA. The trajectory data have been transcribed for every vehicle from the footage of video cameras using NGVIDEO. Among other attributes, each data point has a timestamp, vehicle ID, local road coordinates, global coordinates, vehicle length, width, velocity and acceleration. This dataset consists of more than 11,800,000+ points.

Because of the memory requirements of these algorithms and the relatively small amount of GPU memory available, we ran the algorithms using random subsets of each of these datasets. The sizes of these datasets were in the range from 10,000 to 60,000, as specified in Table 3.1. For this same reason, we kept only the latitude and longitude of each point and discarded the other attributes.

Parameter Name	Range of Values	Default Value
minPts	$10 - 1,000$	1,000
eps	$0.005 - 0.08$	0.05
Dataset Size (num. pts.)	$10,000 - 60,000$	16,384
GPU Block Size	512	512
Number of threads (multi-threaded CPU)	8	8

Table 3.1: Experiment Parameters

3.3.2 Hardware

All experiments were conducted on a machine equipped with a four-core Intel Core i5 3470 chip running at 3.2 GHz, a GTX 1060 GPU with 6 GB of GDDR5 Memory, and a system RAM of 8 GB. The program was executed and benchmarked on Ubuntu 18.04 with Linux kernel version 4.15.0. All algorithms used CUDA 10, Thrust v1.8.1 [48] and were compiled with `-O2` optimizations.

3.3.3 Parameters

All the GPU-based algorithms share, besides *minPts* and *eps*, one parameter in common: the number of threads per GPU block. The size of the GPU grid is calculated automatically from the dataset points and the GPU block size. We fixed the number of threads per block to 512, which is less than the maximum number supported by our GPU.

Unlike the other GPU algorithms, CUDA-DClust has other parameters in addition to the ones already mentioned. There is an index segment size, which is used for indexing but no heuristic is offered for an optimum value. There is also a *numSeeds* parameter, which directly controls the size of the CUDA grid. Again, no heuristic is provided to choose a good value for these parameters. So we set the size of *numSeeds*

to be equal to the number of blocks, in order to ensure that the number of chains increases when increasing the number of points in the dataset.

The range of values used as well as default values are presented in Table 3.1. The range of $minPts$ and eps was selected to produce clusters that vary in size and numbers. Lower $minPts$ value results in fewer, bigger clusters and vice versa. Similarly, lower eps value results in smaller clusters in high quantities and produces more outliers, and vice versa. The default value was chosen to minimize outliers and have most of the data points classified. The dataset size was chosen to be as big as our hardware DRAM could contain and process. Similarly, GPU block size and CPU threads were chosen to be the size on which our hardware performs best.

All approaches were tested for correctness by first running all the GPU and CPU-based algorithms, and then comparing the cluster assignment for each point in the dataset against the clustering of the dbscan package in R [21]. All the algorithms produced the same clustering, i.e., the same assignment from points to clusters.

3.4 Experimental Results

In this section, we present an experimental evaluation of the competing techniques detailed in Section 3.2. We run these techniques on the datasets mentioned in Section 3.3.1 with varying DBSCAN parameters such as $minPts$, eps and number of points in dataset.

3.4.1 Impact of $minPts$

To investigate the impact of the $minPts$ parameter on the execution times of the algorithms, we experimentally found an eps value and a range for the $minPts$ variable where the clustering changes from a few, large clusters to many, small clusters, which

in turn results in a greater number of outliers. For the 3D Spatial Network dataset, an *eps* value of 0.08 and a $minPts$ between 10 and 1,000 gave the desired results. This variability in both the size and the number of clusters helps simplify the analysis of the various properties of the algorithms tested.

We observe in Figure 3.2(a), Figure 3.2(b) and Figure 3.2(c) that with the three datasets, all the algorithms exhibit little to no change in their execution times as the $minPts$ parameter varies across its range.

We also notice in those figures that the single-threaded CPU algorithm shows execution times that are at least 2.5X slower than the algorithm with the closest execution times, which is the algorithm of Thapa et al. for the Porto and Spatial datasets, and the multi-threaded CPU algorithm for the NGSI dataset. In particular, the single-threaded CPU algorithm is on average, across all datasets, 6.35X slower than the multi-threaded CPU algorithm. Since 8 threads were used in the multi-threaded CPU algorithm, this result is not quite the 8X speedup that would be expected. The reason for this is that the multi-threaded CPU algorithm does not parallelize all sections of the program.

In those same figures we see that G-DBSCAN exhibits the shortest execution times in the Porto and 3D Spatial datasets, while lagging closely behind CUDA-DClust in the NGSI dataset. Both have extremely close execution times to those of G-DBSCAN average of 61.7195 ms, 31.9880 ms, and 289.0114 ms on Porto, 3D Spatial, and NGSI, respectively, while CUDA-DClust taking 142.3433 ms, 138.4204 ms, and 266.8608 ms on the corresponding datasets on average.

The multi-threaded CPU algorithm has shorter execution times than Thapa et al.'s algorithm on two of the three datasets tested, Porto Taxi and 3D Spatial, while on the NGSI dataset, Thapa et al.'s algorithm has shorter execution times than the multi-threaded CPU algorithm. An interesting trend is the slight decrease in the

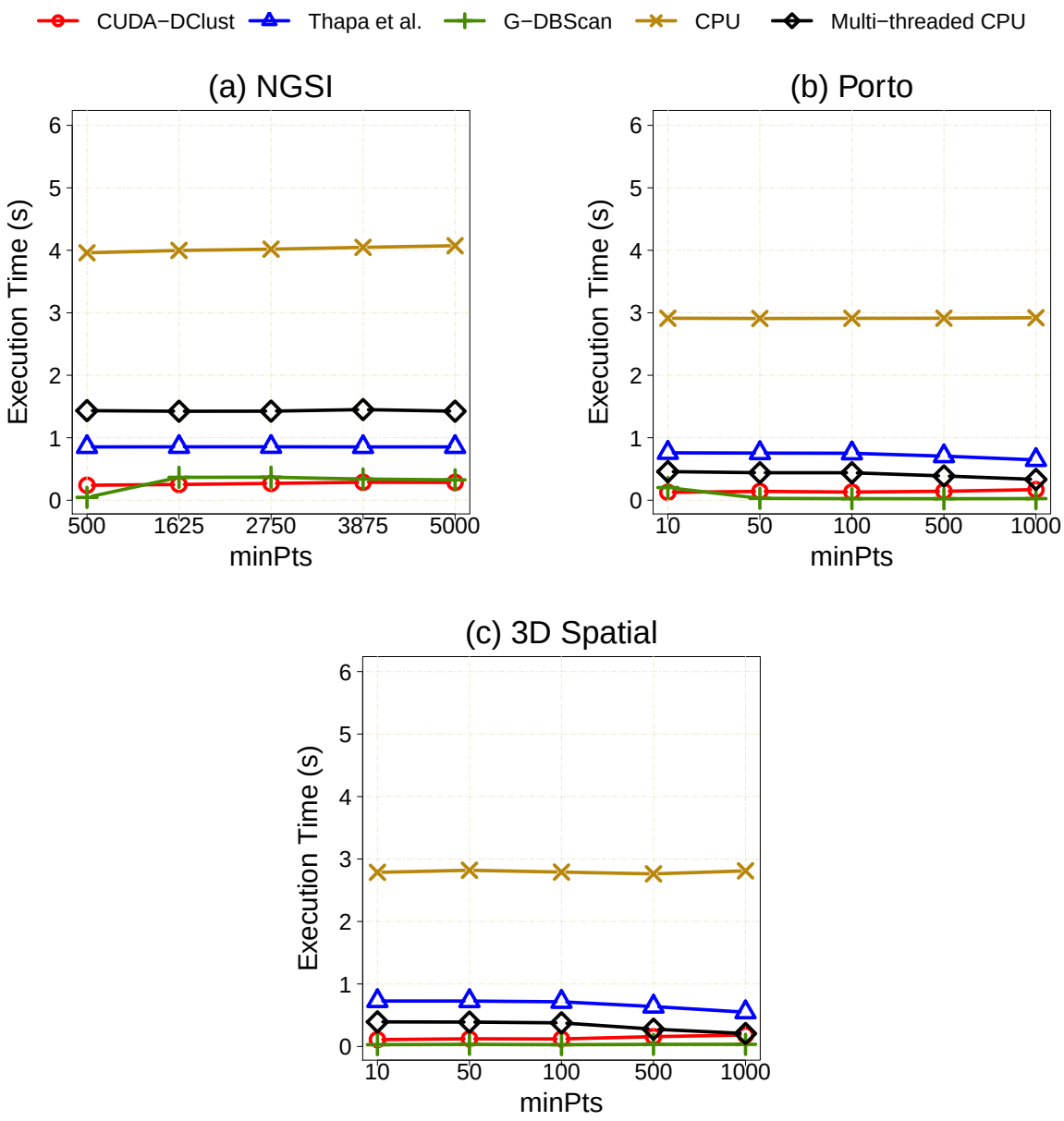

Figure 3.2: Impact of *minPts* on execution time.

execution times of these algorithms when *minPts* increases. With the Porto dataset, DBSCAN finds 5 clusters when *minPts* is 10 (with 16,230 of the total 16,384 points assigned to a single cluster) and 1 cluster when *minPts* is 1,000 (with 10,939 points in that single cluster).

If we look closely in the same figures, the average execution times of G-DBScan and Thapa et al. barely decrease with larger $minPts$ values, whereas the average execution time of CUDA-DClust increases with larger values of $minPts$. This is directly correlated with the number of clusters that a dataset has at those particular parameters. With larger $minPts$ values, DBSCAN finds fewer clusters in the dataset, and many more points are classified as outliers. This works well for G-DBSCAN since it creates a smaller adjacency list that is even faster to traverse. CUDA-DClust on the other hand, keeps switching between seed points for chains, for all except few seed points result in clusters.

3.4.2 Impact of *eps*

The goal of this experiment is to determine the nature of the impact of the *eps* parameter on the execution times of the algorithms. To this end, we empirically found a value for the $minPts$ parameter and a range of values for the *eps* parameter that force the algorithms to output a wide range of clusters in terms of their sizes and numbers. The parameters values $minPts = 100$ and *eps* between 0.01 and 0.1 accomplish this objective for the 3D Spatial dataset.

As we can observe in Figure 3.3(a), Figure 3.3(b) and Figure 3.3(c), the single-threaded CPU algorithm remains the least efficient throughout these tests. Its execution time is on average 4X slower than the second slowest technique at best and 78X slower than the fastest technique at worst. There is an increase in execution time as the value of *eps* is increased because larger *eps* values lead to larger clusters. This increase in the cluster sizes puts a greater load on the *ExpandCluster* function, which then has to label more points to make a larger cluster. The same effect can be seen in the multi-threaded CPU algorithm albeit on a smaller scale.

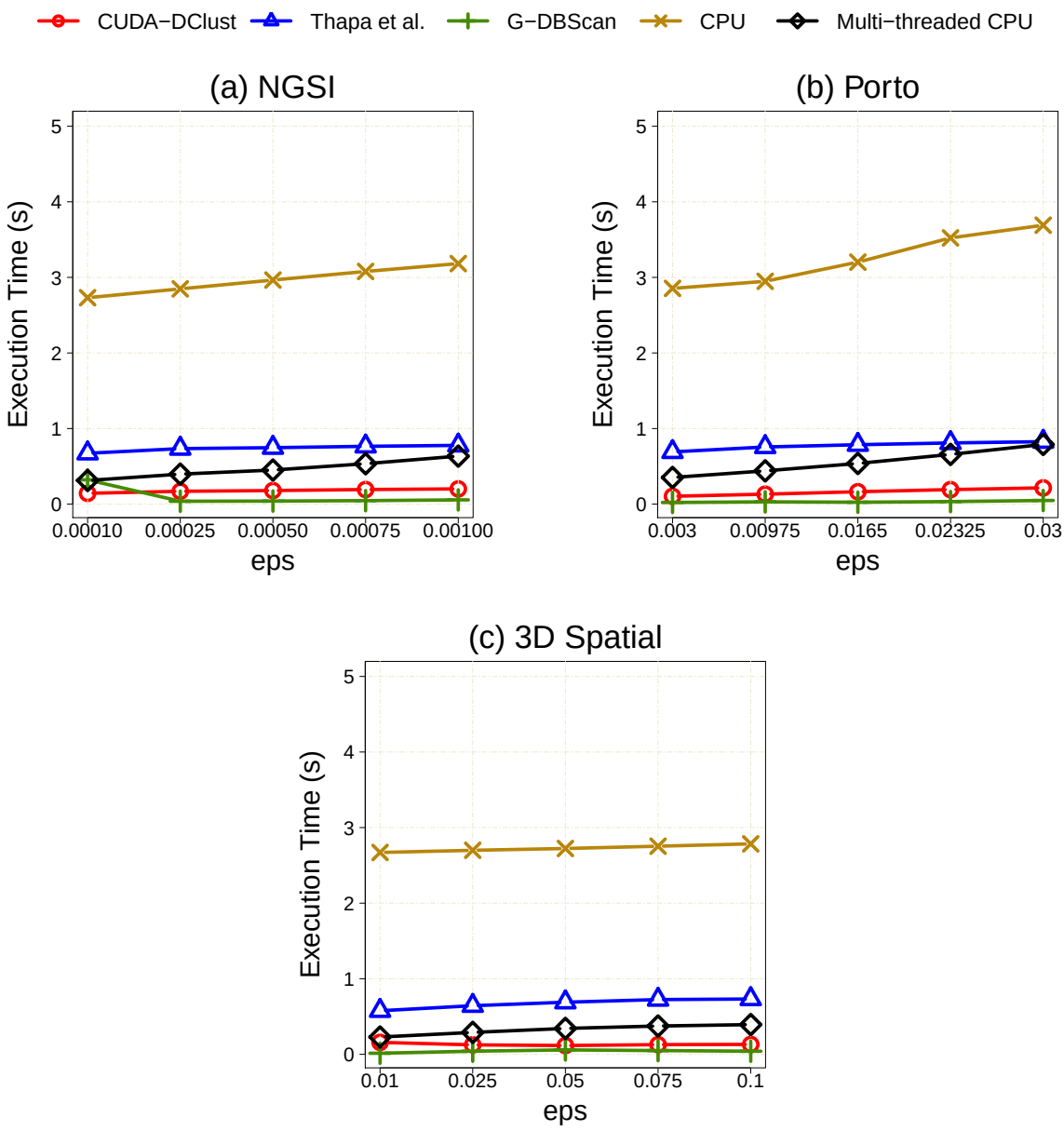

Figure 3.3: Impact of *eps* on execution time.

In these figures we also see that the multi-threaded CPU algorithm has shorter execution times than Thapa et al.'s algorithm on all the datasets. The average execution times of the multi-threaded CPU and Thapa et al.'s algorithm are 555.4 ms and 773.66 ms on Porto, 325.4 ms and 672.75 ms on 3D Spatial and 465.93 ms and

739.01 ms on NGSI, respectively.

Although almost imperceptible, the execution times of G-DBSCAN and CUDA-DClust increase on average with larger values of *eps*. In the Porto dataset for example, the time taken for both algorithms nearly doubles during the span for the tested radius. This means that many, small clusters are easier to process for both these algorithms than few, large clusters. For example, at *eps* equal to 0.003 there are 11 clusters in the dataset, which takes G-DBSCAN and CUDA-DClust 21.8 ms and 103.7 ms, respectively. At *eps* equal to 0.03, however, there is just one, big cluster, which causes an increase in the processing time to 47.9 ms and 214.8 ms, respectively, both values almost twice as long.

3.4.3 Impact of the Number of Points in the Dataset

All GPU-based DBSCAN algorithms should scale in terms of their execution times when the number of points increases because this can help the GPU achieve maximum occupancy. We stress-tested all the algorithms with the dataset points.

Since we increase the number of points in the dataset in an exponential manner, all the plots in this section are in logarithmic scale. As can be seen in Figure 3.4(a), Figure 3.4(b) and Figure 3.4(c), and in the other experiments in this paper, the single-threaded CPU DBSCAN algorithm with R-tree indexing was the slowest out of all the algorithms tested. The disparity is significant, going as far as being 969X slower than the fastest algorithm, G-DBSCAN, on the 3D Spatial dataset with 128,000 points.

The best-performing algorithm in this set of experiments is G-DBSCAN. However, in Figure 3.4(b) corresponding to the Porto taxi dataset, it can be seen that there is no execution time plotted for G-DBSCAN for 131,072 points, because this algorithm could not process more than 65,536 points. The reason for this is that for

131,072 points, G-DBSCAN requires more global memory than the 6 GB available on our GPU. Moreover, the apparent speed advantage that the G-DBSCAN algorithm provided in the previous tests faded when the size of the adjacency list increased, as can be seen in the Porto dataset plot. However, it is still the fastest algorithm, outperforming the second best-performing algorithm, CUDA-DClust, by an order of magnitude: 9X faster in the NGSI dataset, and 18X faster in the 3D Spatial dataset when using 131,072 points.

CUDA-DClust is the overall winner in terms of processing time and memory overhead, taking just as much memory as Thapa et al.'s algorithm, single-threaded CPU, and multi-threaded CPU, and outperforming all but G-DBSCAN in terms of execution time.

As can be seen in Figure 3.4(a), Figure 3.4(b) and Figure 3.4(c), Thapa et al.'s algorithm and the multi-threaded CPU algorithm were really close to each other in terms of execution time, which is unexpected because despite that they follow the same parallelization strategy, one of the algorithms uses GPUs while the other does not. In the Porto dataset, Thapa et al.'s algorithm was faster overall than the multi-threaded CPU, but in the NGSI and 3D Spatial datasets, the multi-threaded CPU algorithm performed the best out of these two algorithms. Thapa et al.'s algorithm had the advantage of having thousands of parallel CUDA threads that could divide the load better when the number of points increased, while the multi-threaded CPU algorithm ran on only eight CPU threads. Nonetheless, the Thapa et al. algorithm has significant overhead stemming from the many kernel launches that it performs. Additionally, Thapa et al.'s algorithm had to deal with the additional overhead stemming from the memory transfers to and from the GPU's RAM to the host's RAM. These reasons help explain why Thapa et al.'s algorithm did not outperform the multi-threaded CPU by a larger margin.

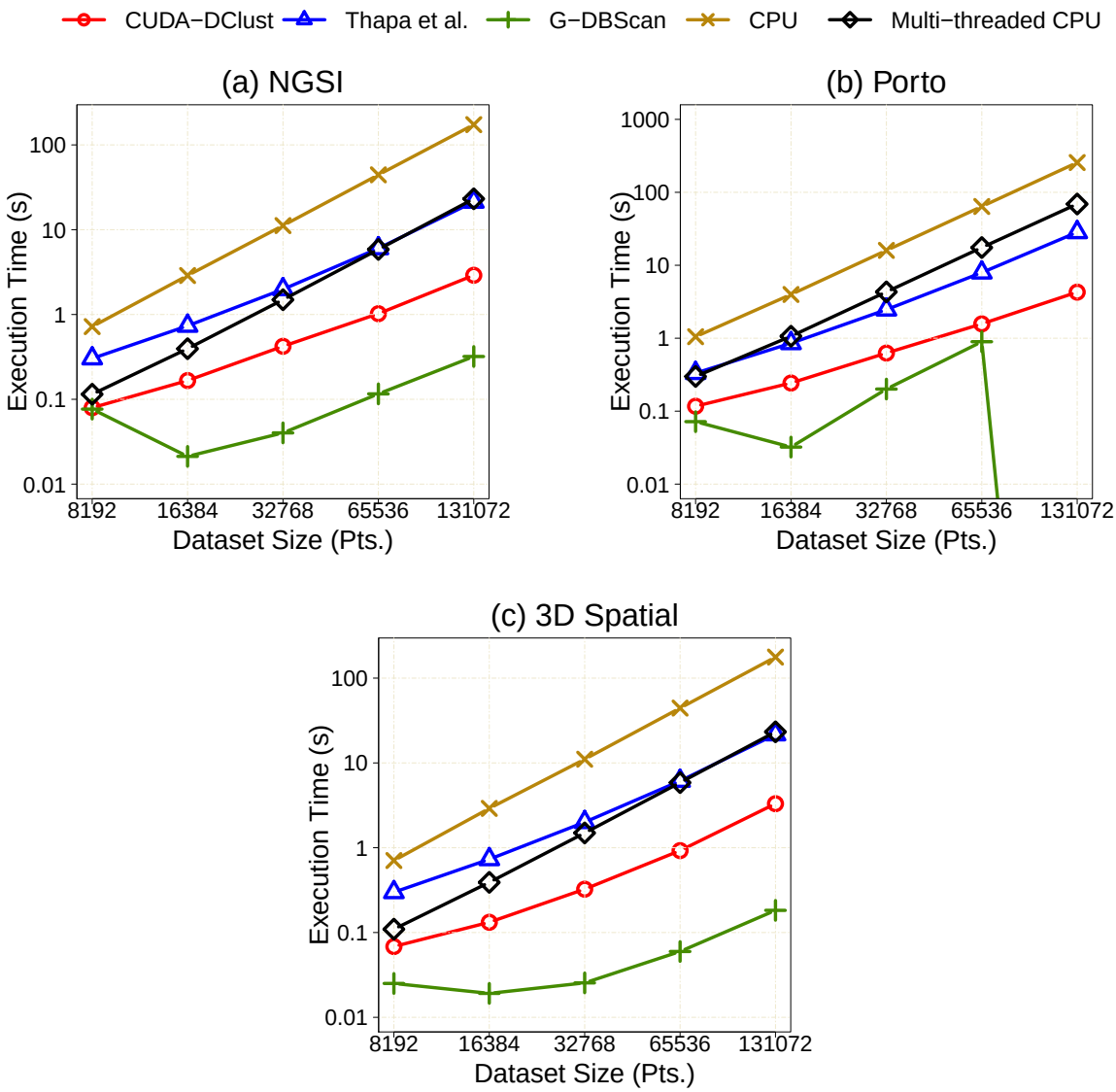

Figure 3.4: Impact of dataset size on execution time.

3.4.4 Memory Impact

For the Porto Taxi dataset in Figure 3.5, the amount of RAM consumed by the algorithms was plotted on a logarithmic scale, since the difference in memory consumption between G-DBSCAN and the other algorithms was too large to plot it with a linear scale. At 32,768 points, G-DBSCAN requires 2.6 GB of RAM, which

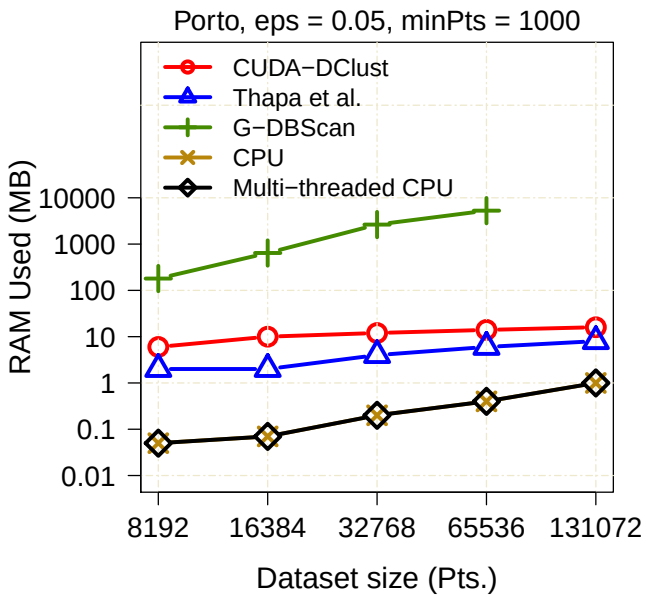

Figure 3.5: Impact of the dataset size on GPU RAM consumption.

is around 166X the amount of RAM required by the CUDA-DClust. The primary reason for this large amount of RAM is that this algorithm stored in the GPU's global memory the complete adjacency list for the density connectedness graph, which for 32,768 points has 690,055,312 edges, each edge consisting of integers of 4 bytes each.

The second worst algorithm, by a significant margin, is CUDA-DClust, which required around 16 MB for 131,072 points. Most of this memory was consumed because of the indexing data structures and a few data structures to store seed expansions. Thapa et al.'s algorithm came third with a consumption of 8 MB for 131,072 points. Since there were no additional variables used, the expected memory consumption should have been even lower, but some housekeeping data at the thread level in GPU made it to be multiple MBs.

Both the single-threaded and the multicore CPU algorithms used roughly the same amount of memory. This is expected since the structures of both algorithms are very similar and there are no auxiliary data structures initialization besides the

label for each point. Thus, the memory consumed was a couple of KBs at best and barely reached one MB in the worst case.

3.5 Discussion

In this section we provide a brief discussion on G-DBSCAN [3], CUDA-DClust [6], Thapa et al.'s algorithm [59], and multi-threaded CPU DBSCAN [13] and briefly examine the strengths and weaknesses of each.

- The disadvantage with G-DBSCAN is apparent. Because it allocates the adjacency list for the density connectedness graph in the GPU's RAM, it requires large amounts of global memory. However, once the list is created, the process of finding clusters using breadth-first search is fast and efficient to the point that in most cases it is able to outperform the other algorithms. However, stress tests revealed that the GPU BFS in G-DBSCAN tends to slow down as the adjacency list grows larger, which it does for big datasets; for this reason, G-DBSCAN has difficulty scaling beyond the global memory capacity of GPUs. Thus, even having unlimited memory cannot fix the issues it brings. In our tests however, it proved to be the fastest choice, performing up to 18X faster than the closest one, CUDA-DClust, and up to 969X faster than CPU DBSCAN.

- The only issue with CUDA-DClust is that it is an algorithm that is difficult to implement. The code provided to us by Böhm et al. [6] contains several enhancements, which include a dynamic load-balancing technique that ensures that the GPU is always occupied and working on expanding multiple chains. There is a very simple but effective indexing technique used as well as careful implementation of some atomic operations where the mentioned chains conflict.

The result of this is a well-rounded, fast, and memory efficient algorithm that adequately utilizes the GPU.

- Thapa et al.'s algorithm makes every thread on the GPU perform only one distance calculation; this underutilizes the potential of each thread while inducing a significant kernel launch overhead since the process is repeated inside a for loop for every point. So for every point, the kernel launch overhead adds up for a brief, quick distance calculation at each thread with a memory transfer overhead following it. This is the reason why only 4 CPU threads outperform this algorithm for smaller datasets because they do not have this overhead at every iteration of the loop. These issues can obviously be eliminated by running the kernel once where every thread processes all the distances, but doing this requires $O(n^2)$ memory, an amount that not even G-DBSCAN might reach (it gets close though when the points are really close to each other like in the Porto dataset). So going that route, the algorithm will probably fail with an input with a size of roughly 40,000 points, assuming a GPU with a capacity of 6 GBs.

- The multi-threaded CPU algorithm is appropriate to use in the absence of a coprocessor or when dealing with smaller datasets. The independence of each distance calculation can be easily exploited by different cores of a CPU. However, CPU cores are usually orders of magnitude times fewer in number than the number of CUDA cores in a typical GPU, so the only issue with the multi-threaded CPU algorithm is that it does not scale as well as the GPU algorithms for larger datasets. This problem is shared with the original single-threaded CPU DBSCAN with indexing too, which as our tests proved, is not suitable for clustering if processing time is of any importance.

4 GTRACLUS: A GPU-based Trajectory Clustering Algorithm

In Chapter 3, we experimentally found out that G-DBSCAN performs the fastest among all the GPU-based DBSCAN algorithms while CUDA-DClust was the best all around. Following that, we apply our findings to solve the problem with trajectory clustering algorithm, TRACLUS [37]. Due to the quadratic time complexity and increased size of real-world datasets, TRACLUS performs the clustering of trajectories using DBSCAN [13] in a relatively high amount of time. In this chapter we present GTRACLUS, a novel GPU-based trajectory clustering algorithm that performs more than 20X faster than TRACLUS. GTRACLUS replaces DBSCAN in TRACLUS with a modified version of G-DBSCAN built for clustering trajectories. Moreover, as provided in this chapter, we use additional techniques to speed up GTRACLUS on GPU in every stage of this algorithm. Section 4.1 provides related work. Section 4.2 details the proposed GTRACLUS algorithm in detail and Sections 4.3 and 4.4 present the experimental evaluation of TRACLUS and GTRACLUS on two real-world trajectory datasets.

4.1 Related Work

4.1.1 Trajectory Clustering

As mentioned in Chapter 1, the data mining community has to deal with unprecedented amount of multi-variable time-series data partly because of the plethora of easily attainable GPS devices and cheap storage space. This provides a great opportunity for analysis of similar patterns on time varying data by clustering the trajectories into groups containing similar trajectories. This has a broad range of applications in migration pattern-based bird identification, location-based social networks [64], recommendation of travel locations of interest based on common trajectories [69], finding users with similar life experiences based on their trajectories [39], intelligent transportation systems, and urban computing [67], drug therapy monitoring, gene expression protein modeling and individual responses stimuli in animal behavior experiments. Trajectory clustering can also be employed in epidemiology, to help centers of disease control contain the spread of avian influenza by discovering the underlying migration trajectories of mallards [25]. It can also be used in trajectory-based advertising, where a shopping mall, after tracking the movements of the shoppers that have logged into the mall's wifi network, can send personalized advertising information to customers based on their paths inside the mall [17].

Due to the importance of its application, clustering of trajectories has been an active research topic since as early as 1999 [15]. Several works explore trajectory clustering, as well as other operations in detail [65]. Gaffney et al. [15] in 1999 proposed a trajectory clustering algorithm which is probabilistic mixture regression model-based and applied EM (Expectation-maximization) to cluster trajectories. Evaluation was performed on 20 video streams depicting 5 different hand movements to cluster the two-dimensional time series trajectories based on hand motion with success (100% ac-

curacy). Gaffney and Robertson et al [14] later performed curve-based mixture modeling techniques to perform clustering of Wintertime extratropical cyclones (ETC) to gain insights into weather and climate prediction. A novel probabilistic technique based on mixture of regression models was used.

Lee et al. [37] noticed that the mentioned algorithms do not account of sub-trajectories within trajectory data. Clustering of trajectories based on their overall distance may not provide important insights into the common shorter paths the objects took as real-world objects don't always take a similar path for the entirety of their journey. In the two examples provided, they mention that while predicting hurricane landfall based on hurricane trajectories, meteorologists are more interested in clustering hurricane behaviors near the coastline or at the sea rather than performing clustering operation on their entire hurricane trajectories. Similarly, while examining effects of vehicular traffic on animal movement, distribution, and habitat use, zoologists are more interested in common behaviors of animal trajectories near the road. To this end Lee et al. proposed TRACLUS [37], a framework to break trajectories into a set of line segments and find clusters among them. It also provides a method of determining a representative path for the clusters of line segments.

This operation however, is computationally intensive. In our results, one TRACLUS computation on a real-world dataset took up to 40 minutes (without representative path evaluation). The reason for this is DBSCAN clustering for line segments, which is used in the clustering stage of the algorithm, requires evaluation of three kinds of distances between one line segment to every other segment in the dataset. This makes it $O(n^2)$ complexity where n is the number of line segments.

Lee et al. also created an [40] Incremental algorithm for trajectory clustering, TCMM, that does online and offline processing for non-static datasets incrementally as they arrive. The algorithm creates and stores micro-clusters every time a

new trajectory arrives which is space efficient. Later, macro-clustering is performed on micro-clusters rather than on all trajectories. Since there is an information loss while clustering representative micro-clusters from data, the accuracy of the results is negatively affected. Thus, while experimentally the processing time is faster than TRACLUS, the accuracy reduces to 2% SSQ (sum of square distance). Moreover, this algorithm also adds another parameter *dmax* to the micro-clustering step and there is no heuristic to select an optimum value for it. So even though it is a much faster trajectory clustering algorithm than TRACLUS, it trades accuracy for speed and adds an extra parameter. Hence, there is still a need for a accurate algorithm that performs trajectory clustering more efficiently.

Roh et al. also proposed a trajectory clustering algorithm NNCluster [56] that performs clustering on road-network trajectories specifically. They propose a new, order dependent distance measure for traffic trajectories and use a agglomerative hierarchical clustering which checks for common nearest neighbors instead of a complete dataset search. Not only is this work highly application specific it also doesn't account for common sub-trajectories like TRACLUS does.

Pelekis et al. proposed the algorithm CenTra-I-FCM [52] which works in three steps. First, it represents trajectories in vectors, then it discovers centroids between them using C-Means, and finally it post processes these data by applying fuzzy logic to obtain patterns of uncertainty. As with NNCluster the clustering is performed on trajectories as a whole, which is undesirable for our application. A relative strength of this algorithm is its fuzzy clustering that may assign multiple cluster membership to a single trajectory.

Table 4.1 summarizes all the related work done in the field of trajectory clustering and provides a simple comparison to get the reader familiar with the techniques. The table provides characteristics of all the discussed techniques, including the cluster

Technique	Cluster Method	Distance Measure	Whole or Partitioned	Application	GPU
Gaffney et al. [15]	EM Algorithm for mixtures of linear regression models	Mahalonobis distance for Gaussian clusters (any appropriate metric can be used)	Whole	General Purpose	No
Gaffney and Robertson et al. [14]	Curve-based mixture modeling techniques to perform probabilistic clustering	Mahalonobis distance for Gaussian clusters (any appropriate metric can be used)	Whole	ExtraTropical Cyclone Trajectories	No
TRACLUS [37]	DBSCAN with multiple distance measures	Proposed distance functions; *perpendicular distance, parallel distance* and *angle distance*	MDL-based partitioning	General Purpose	No
TCMM [40]	Clustering on micro clusters of incoming trajectories converted into line segments	Proposed distance functions; *center point distance, parallel distance* and *angle distance*	Partitioned (Provided at input)	Streaming Trajectories	No
NNCluster [56]	Approximate agglomerative hierarchical clustering of k nearest neighbors done using a novel order-dependent distance measure	Proposed order-dependent distance measure	Whole	Road Network Trajectories	No
CenTR-I-FCM [52]	Fuzzy C Means Clustering	Proposed novel distance metric which is a modification of Edit distance	Whole	General Purpose	No
GTRACLUS	DBSCAN with multiple distance measures	*Perpendicular distance, parallel distance* and *angle distance*	MDL-based partitioning	General Purpose	Yes

Table 4.1: Comparison of Trajectory Clustering Techniques.

method, which varies among the techniques. It also mentions GTRACLUS, the technique proposed in this chapter. The distance function used is specified, which often is a novel strategy proposed for the technique. It also specifies whether the entire trajectory is clustered or partitioning taking place. Finally Table 4.1 also mentions the intended application by the authors of the corresponding techniques, and whether or not these run on GPUs.

4.1.2 TRACLUS

TRACLUS [37] is a trajectory clustering algorithm that partitions and groups trajectories into clusters. The partitioning part is important because multiple trajectories may have a subset of points that form a similar path but their overall distance may be too great to be clustered into one group. This clustering of sub-trajectories has applications in animal movement analysis and hurricane landfall forecasts. Lee et al. provide a counter-argument to this approach that useless parts of trajectories can be discarded and only useful sub-trajectories can be used for clustering. However, they further explain that determination of usefulness may be difficult and also discarding data may result in lack of discovery of unexpected results.

To that end, TRACLUS provides a two-phase evaluation of trajectory clustering: partitioning and grouping. Given a trajectory, partition phase outputs a set of line segments based on MDL (Minimum Description Length) principle. And grouping stage uses density-based clustering to cluster common line segments in the same group. The clustering of the line segments take place using a density-based clustering algorithm called DBSCAN. DBSCAN is used here because of its general strengths like resilience against noise and ability to discover clusters of arbitrary shapes. Historically, K-means hasn't been used to cluster trajectory in other works in trajectory

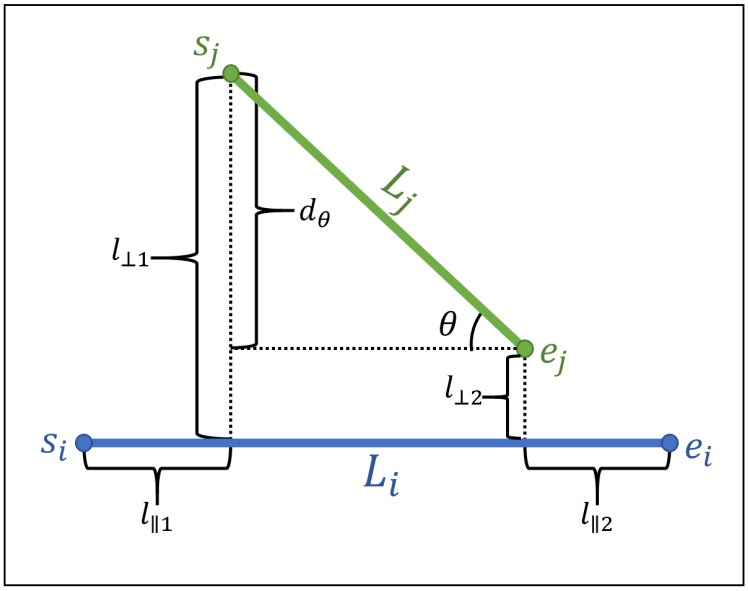

Figure 4.1: Components of the distance function in TRACLUS.

clustering mentioned in Section 4.1.1 because it can't be applied to data of different lengths. To apply K-means the trajectories must be converted to fixed-dimensional vectors that may lose spatio-temporal information.

Given a set of trajectories $I = TR_1, ..., TR_{num_{tra}}$, the algorithm generates a set of clusters $O = C_1, .., C_{num_{clus}}$ as well as a representative trajectory for each cluster C_i. This operation is performed by TRACLUS in two phases explained in greater detail as follows.

Distance Measures

The distance measures used in TRACLUS comprise of three different functions, the *perpendicular distance*($d_\perp$), the *parallel distance*($d_\parallel$), and the *angle distance*(d_θ). Here we briefly define these distances. On d-dimensional line segments, $L_i = s_i e_i$ and $L_j = s_j e_j$, where L_j is the shorter line segment, illustrated in Figure 4.1, these distances would be as follows.

- The *perpendicular distance* between L_i and L_j is:

$$d_{\perp}(L_i, L_j) = \frac{l_{\perp 1}^2 + l_{\perp 2}^2}{l_{\perp 1} + l_{\perp 2}}$$

where $l_{\perp 1}$ is the Euclidean distance between s_j and projection of s_j on L_i, and $l_{\perp 2}$ is the Euclidean distance between e_j and projection of e_j on L_i, as shown in Figure 4.1.

- The *parallel distance* between L_i and L_j is:

$$d_{\parallel}(L_i, L_j) = MIN(l_{\parallel 1}, l_{\parallel 2})$$

where $l_{\parallel 1}$ is the minimum of Euclidean distances of projection of s_j onto L_i to s_i and e_i. Similarly, $l_{\parallel 2}$ is the minimum of Euclidean distances between projection of e_j onto L_i to s_i and e_i. $l_{\parallel 1}$ and $l_{\parallel 2}$ are shown in Figure 4.1 for lines L_i and L_j.

- The *angle distance* between L_i and L_j is:

$$d_{\theta}(L_i, L_j) = \begin{cases} ||L_j|| \times \sin(\theta), & \text{if } 0° \leq \theta < 90°. \\ ||L_j||, & \text{if } 90° \leq \theta \leq 180°. \end{cases}$$

where $||L_j||$ is the length of L_j and θ is the smaller intersecting angle between L_i and L_j. d_{θ} is shown in Figure 4.1 for lines L_i and L_j.

Partitioning

From every trajectory, this stage determines *characteristic points*, the points where the behavior of trajectory changes rapidly. All the partitioned line segments start and

end between two consecutive characteristic points. These line segments constitute the partitioned trajectories.

Trajectory partitioning into line segments has two desirable properties: preciseness and conciseness. Preciseness is the difference between a trajectory and a set of its partitioned line segments. Conciseness is minimizing the number of line segments needed to represent the trajectory. Both these properties require increasing ideally, however growing one diminishes the other. Thus, an optimal balance is needed between the two.

Lee et al. propose MDL principle as the solution to this optimization between preciseness and conciseness. The MDL principle introduced by Rissanen [55] is an important concept in information theory and is a formalization of Occam's Razor. MDL principle states that if data D with length $L(D)$ is encoded using a hypothesis H whose length is $L(H)$ and the compressed data after encoding with hypothesis H is $L(D|H)$ then $L(D) = \min(L(H) + L(D|H))$. Which essentially means that the hypothesis used to describe data will be the best one if it can compress it the most as well as the hypothesis itself can be encoded with minimum length.

With a trajectory $TR_i = p_1p_2p_3...p_j...p_{len_i}$ and a set of characteristic points $= \{p_{c_1}, p_{c_2}, p_{c_3}, ..., p_{c_{par_i}}\}$, Lee et al. formulate $L(H)$ and $L(D|H)$ as follows:

$$L(H) = \sum_{j=1}^{par_i-1} \log_2(len(p_{c_j}p_{c_{j+1}}))$$

$$L(D|H) = \sum_{j=1}^{par_i-1} \sum_{k=c_j}^{c_{j+1}-1} \{\log_2(d_\perp(p_{c_j}p_{c_{j+1}}, p_kp_{k+1})) + \log_2(d_\theta(p_{c_j}p_{c_{j+1}}, p_kp_{k+1}))\}$$

The $L(H)$ is the log of $len(p_{c_j}p_{c_{j+1}})$ which essentially the length of a line segment

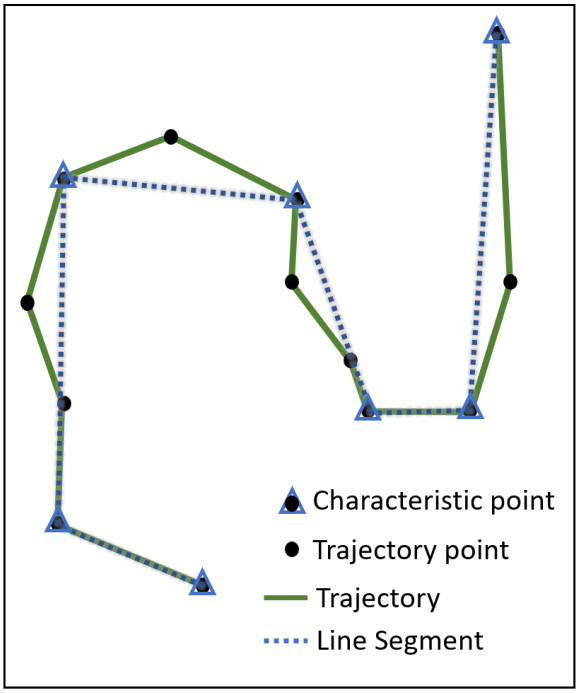

Figure 4.2: The partition process of trajectory to line segment done by TRACLUS.

$p_{c_j}p_{c_{j+1}}$ which is the Euclidean Distance between the points p_{c_j} and $p_{c_{j+1}}$. The $L(D|H)$ is the sum of the difference between a trajectory point and its trajectory partitions. The difference is calculated using the sum of perpendicular and angular distance.

Lee et al. note that $L(H)$ measures conciseness and $L(D|H)$ measures preciseness. To minimize $L(H) + L(D|H)$, TRACLUS uses an approximate solution that regards local optima as the global optima. If MDL cost between two characteristic points p_i and p_j is $MDL_{par}(p_i, p_j)$ and the MDL cost when there is no characteristic point between p_i and p_j is $MDL_{nopar}(p_i, p_j)$ then local optima is the longest trajectory partition p_ip_j that satisfies $MDL_{par}(p_i, p_k) \leq MDL_{nopar}(p_i, p_k)$ for every k such that $i < k \leq j$.

Thus, to partition a trajectory into line segments, the MDL_{par} and MDL_{nopar} costs are calculated for each point in the trajectory and if MDL_{par} comes out greater than MDL_{nopar} the previous point is added into the set of characteristic points.

These characteristic points together make line segments that are sub-trajectories of a trajectory and sent to the next step of TRACLUS to be clustered. Figure 4.2 shows the process of splitting a trajectory into line segments based on characteristic points. At every trajectory point where MDL_{par} is greater than MDL_{nopar} a characteristic point is created, shown as a triangle in Figure 4.2. Then from these characteristic points individual line segments are created shown, as dotted line in the figure. The heuristic used by Lee et al. for MDL calculation strives to be as precise as possible, that is not losing important information from trajectory data, which also being as concise as possible, that is compressing the original information optimally.

One other similar technique that can be used to calculate MDL is DynMDL algorithm [34]. However, in their experiments Leal et al. concluded that even though DynMDL produces lower description length (better compression) it performs slower than MDL. Since improved performance in terms of speed is our goal, we will use the original MDL principle for our work as proposed in TRACLUS.

Grouping

The grouping state is where the clustering takes place, using density-based clustering algorithm DBSCAN [13]. But unlike the original DBSCAN, here it has been modified to utilize the distance measures described earlier to cluster line segments instead of points.

The distance measure, *dist*, used for clustering is a weighted sum of *perpendicular distance, parallel distance* and *angle distance*. The *epsilon neighborhood* is redefined here for line segments as $\{ L_j \in \mathcal{D} \mid dist(L_i, L_j) \leq \epsilon \}$ where $\mathcal{D}$ is the set of all line segments. A *core line segment* is defined as one which has more than *minLines* number of line segments in its epsilon-neighborhood.

Figure 4.3 shows the clustering process of line segments. All colored line segments,

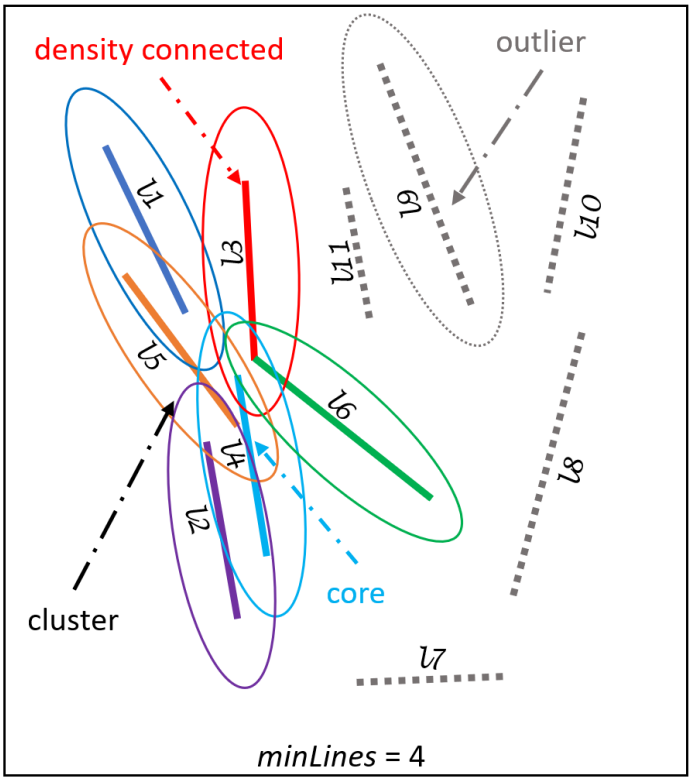

Figure 4.3: The clustering process of line segments done by TRACLUS.

l1-l6, form a cluster, whereas all gray, dotted line segments are outliers. The ellipses around line segments specify their epsilon-neighborhoods. Since line segment l4 has four other segments in its epsilon-neighborhood and *minLines* is 4, it is classified as a core line segment. Line segments l1,l2,l3,l5 and l6 form part of the cluster even though they do not have *minLines* number of lines in their epsilon-neighborhoods. This is because they are density-reachable from the core line segments. Lines l7−l11 form outliers since they neither have *minLines* number of lines in their neighborhood, nor are density connected to such a line segment.

Unlike DBSCAN, not all density connected line segments can become clusters. Line segments that form a cluster, if coming from the same trajectory do not provide us any interesting information. Hence the concept of *trajectory cardinality* is used

which is the number of trajectories a cluster has instead of number of line segments.

Other than this detail, the clustering algorithm works very similar to DBSCAN. All line segments are initially unclassified. The algorithm starts by determining the epsilon-neighborhood of each line segment. If the line segment is classified as a core point by checking its epsilon neighbors against *minLines* variable, the `expandCluster` routine is executed which expands a cluster from the core point. All line segments not classified with membership of a cluster are labeled as noise. Lastly, the algorithm checks for trajectory cardinality for each cluster, and discards the clusters with trajectory cardinality below a certain threshold. The output is a set of partitioned and labeled trajectories.

4.2 Proposed Algorithm

In this section, we present a new parallel trajectory clustering algorithm called GTRACLUS.

4.2.1 Overview

The grouping stage of TRACLUS algorithm is where it takes the most number of computation operations since it is the bulk of the clustering procedure. In our tests, the average amount of time spent on the grouping stage across all trajectory sizes, epsilon and minLines was 99.037% for Porto Taxi dataset and 99.93% on Geolife dataset. So this area was the first in priority while making TRACLUS more efficient.

However, for fewer trajectories (≈ 100) the partitioning time dominates. TRACLUS spends more time creating line segments than clustering those line segments. In our experiments, the partitioning time was 5 – 10 times longer than the grouping time for small datasets. Even though it was quickly dominated by grouping as the

number of trajectories increased (since partitioning is linear and grouping is quadratic in n on number of lines), still to make TRACLUS faster all around, this stage needs to be optimized too. This will ensure a faster computation time regardless of the number of trajectories and trajectory size.

To that end we propose GTRACLUS, a GPU-based trajectory partition-and-group framework. This new framework combines the findings of our previous research on the best GPU-based DBSCAN algorithm as well as introduces new optimization strategies to bring the computation time of TRACLUS down by orders of magnitude.

Algorithm 4.1: GTRACLUS Algorithm

Input: Dataset D, *epsilon*, *minLines*
Output: Line segments *segments* from trajectories in D and *labels* indicating their cluster ID

```
1  function GTRACLUS(D,epsilon,minLines)
2      segments ← GpuPartition(D);
3      labels ← GDBSCAN(segments,epsilon,minLines);
4      return segments, labels;

6  function GpuPartition(D)
7      for each trajectory t in D do in parallel
8          dSegs[t] ← CountPartitions(t);
9      dSegs ← InclusivePrefixSum(dSegs);
10     for each trajectory t in D do in parallel
11         segments ← FillPartitions(t,dSegs);
12     return segments;
```

4.2.2 Partition

The first stage of GTRACLUS is partition which is done by `GpuPartition` function in line 2 of Algorithm 4.1. The partitioning stage of original TRACLUS was especially challenging to port on parallel architectures because of its sequential nature. MDL_{par} is calculated for every single point in trajectory and compared with

MDL_{nopar} calculated so far from the points seen in that trajectory. Regardless, we employed few techniques that helped us deal with this issue.

The first step was to calculate the number of partitions created by each trajectory. We used `CountPartitions` kernel over the entire dataset, which returned $dSegs$, as shown in lines 7-8 in Algorithm 4.1. The `CountPartitions` assigns a single thread to each trajectory in the dataset whose job is to traverse between characteristic points, calculate MDL cost of that point using perpendicular and angular distance and label it as a characteristic point if the cost is greater than the cost of previous characteristic point. The number of characteristic points determine how many segments a particular trajectory is break into.

Before starting the next step, inclusive scan is computed for number of segments array $dSegs$ returned in the previous step, as shown in line 9 of Algorithm 4.1. All this was done to ensure that all threads in the next step have adequate space to store line segment data without wasting memory. $|dSegs|$amount of memory is allocated on the GPU and the next kernel, `FillPartitions` is run in lines 10-11. This is essentially the same kernel as the `CountPartitions` run earlier but this time it fills up a $2 \times n$ array called $segments$ that stores x and y coordinates of all the segmented points, n. These are the points from which line segments are created. The original dataset can now be safely discarded to conserve GPU memory.

4.2.3 Clustering

For the clustering part, we use a GPU-based DBSCAN algorithm introduced by Andrade et al. called G-DBSCAN [3]. The reasons for this selection is elaborated in Chapter 3 of this thesis.

In Section 3.1.5 we mention that G-DBSCAN creates a density connectedness

graph of all the points that lie within *epsilon* distance of each other. Once the graph is created it performs parallel Breadth First Search (BFS) on the points to find clusters. There are multiple reasons for this strategy being efficient on parallel hardware. One is the extensive amounts of distance calculations between a point to every other point is the type of parallelism where GPU excels, reducing the complexity from $O(V^2)$ for each vertex to $O(V)$. The BFS also utilizes parallel threads to run for every point and scan through its adjacency list in order of breadth first frontiers. This process stays $O(V + E)$ however, every vertex is processed in parallel again. All directly reachable points are labeled as members of the same cluster, while also determining border points of the cluster in case the point does not have $minPts$ neighbors. All the points that are not part of the cluster are labeled as noise.

Section 3.3 compares G-DBSCAN with other GPU-based DBSCAN clustering algorithms in literature and determines that G-DBSCAN performs the fastest. The discussion in Section 3.5 provides some useful insights into this behavior.

For integrating into TRACLUS, tackling a few challenges was in order. The biggest one was G-DBSCAN is built for point data, and TRACLUS performs clustering on line segments. Thus, the G-DBSCAN used to cluster line segments in line 4 of Algorithm 4.1 is similar to Algorithm 3.4 explained in detail earlier, but with the following changes.

The first change was in the `makeGraph` function. Instead of point, all input data types were changed to *lineSegment* structure, which contains x and y coordinates of start point, x and y coordinates of end point, the index of the trajectory the line segment belonged to, and the $clusterID$ of the segment. This information helps us out in the edge finding stage. Like original G-DBSCAN, a slightly modified kernel runs twice. Once to evaluate the starting positions of the adjacency matrix, and a second time to fill out the adjacency matrix. This approach helps us by making

the adjacency matrix compact, and instead of creating space in memory for N^2 line segments, we only allocate memory for the line segments that are needed.

In both `fillNodes` (the adjacency matrix evaluator) and `fillEdges` methods, we replace Euclidean distance calculations between two points with a different distance measure. First, for every line segment we find out the Euclidean distance between its start (sx, sy) and end (ex, ey) points called len_j. Then the thread loops over the entire dataset and for every other line segment the distance between their start and end points is calculated, called len_i. The $totalDistance$ is evaluated by calculating the perpendicular, parallel and angular distance between line segment i and j. The order of these depend on whose length is greater (calculated in the earlier step). The parallel, perpendicular and angular distances require calculation of projection points, euclidean distances and tangent functions. Once calculated, they are added in equal amounts $(1/3^{rds})$ to obtain the total distance t_d. The t_d is compared against epsilon, and if it is greater, the line segment will be connected by a graph, otherwise these line segments are disconnected.

Once the graph is created between the line segments, a `BFS` is initiated, similar to Algorithm 3.4. Since we used an abstraction of the line segments to create graph, the individual vertices in the graphs are their IDs, so graph search can take place unmodified. All core segments (line segments with neighbors greater than $minLines$) become the starting point of the search and each thread goes into the adjacency lists of density connected line segments and marks them as visited. All directly density reachable line segments get the same cluster label, and the process is conducted in parallel. Once all points are marked visited, the algorithm concludes and clustering data is provided as output.

4.3 Performance Analysis

4.3.1 Datasets

For our experiments, we used two real life datasets. Geolife [68] and Taxi Service Trajectory Prediction Challenge dataset [44]. Both these datasets contain a considerable number of time series trajectory data, Geolife having 17,621, which were broken down into 64,000 trajectories and Taxi Service Prediction dataset having 1.7 million. The total number of points in these datasets is more than adequate for our clustering purposes.

4.3.2 Hardware

As with the previous experiments in this thesis, the hardware used was a machine with a four-core Intel Core i5 3470 processor clocking at 3.2 GHz running four threads, 8 GB of RAM and a Nvidia GeForce GTX 1060 GPU with 6 GB of GDDR5 memory.

The program was executed and benchmarked on Windows 10 running CUDA 10.0 and Thrust v1.8.1 [33].

4.3.3 Parameters

Both TRACLUS and GTRACLUS were operated on same parameters for result and time comparison. DBSCAN requires the parameters *epsilon* and *minPoints* to operate. TRACLUS modified the second parameter to *minLines* instead of *minPoints* due to the clustering of line segments instead of points. Increasing *minLines* and decreasing *epsilon* increases the number of clusters and outliers and vice versa. As *minLines* is decreased and epsilon is increased the whole dataset approaches a single, huge cluster. We made sure to employ realistic measurements for both of these,

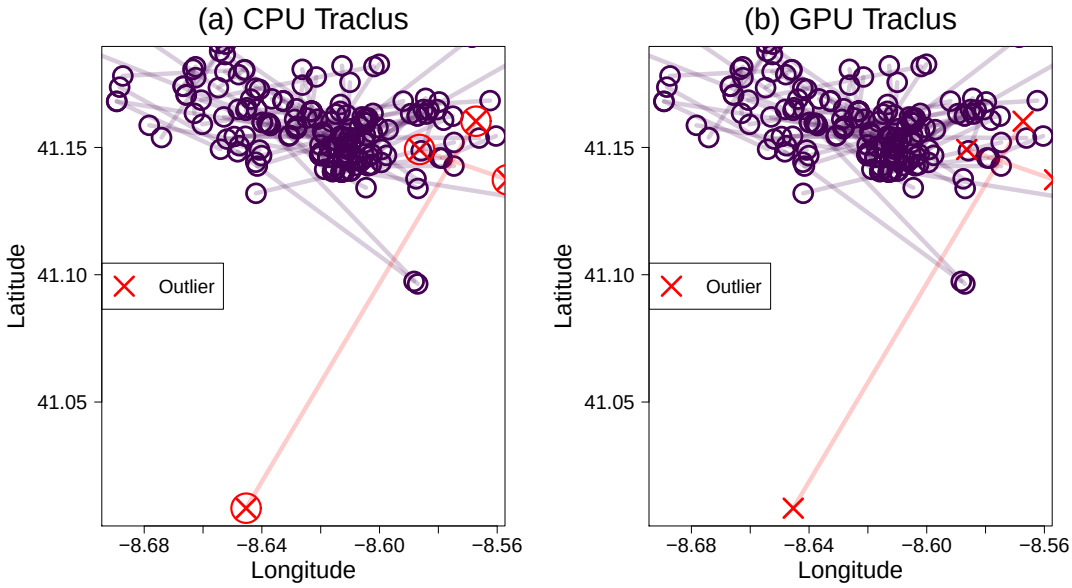

Figure 4.4: Classification of Porto Taxi data line segments.

so as to minimize outliers and have the highest number of lines classified. Since both these datasets contain vehicular coordinates on roads of various cities, the same parameters provided similar behavior. Which is why we conducted the experiments on both datasets using the same parameter values. Table 4.2 provides the parameters we selected for the two datasets.

Parameter Name	**Range of Values**
Epsilon	0.001 – 0.009
MinLines	5 – 100
NumTraj	100 – 100,000
CPU threads (TRACLUS)	4

Table 4.2: Experiment Parameters

4.4 Experimental Results

4.4.1 Correctness

Throughout all the parameter values, GTRACLUS produced the same clustering as TRACLUS. All the line segments got assigned to the same clusters and the outliers were identical. To ensure that TRACLUS is operating correctly (in terms of the parameters provided) we reduced the number of trajectories and created *epsilon* radius ellipses around all the line segments and inspected the intersections of these ellipses. We found out that TRACLUS was operating correctly. Then we repeated the process for GTRACLUS and the results were the same. Thus, the proposed algorithm GTRACLUS is correct in determining line segments into clusters that are within *epsilon* distance and have *minLines* number of line segments intersecting them. Figure 4.4 and Figure 4.5 show the clusters formed on Porto and Geolife datasets, respectively. We can visually see the same line segments being assigned the same cluster groups. Moreover, Figure 4.4 (a) and Figure 4.5 (a) show circles plotted over outlier points to confirm that they indeed have no *minLines* number of segments intersecting them. To further confirm the similarity of their outputs, we ran TRACLUS and GTRACLUS on 100,000 trajectories of both datasets and compared the outputs through a text comparison script. There were zero dissimilarities.

4.4.2 Impact of the Number of Trajectories

Here we report the impact of number of trajectories on the processing time of TRACLUS and GTRACLUS in both the grouping and clustering stages.

Increasing the number of trajectories exponentially altered the processing time of both TRACLUS and GTRACLUS exponentially too. However, the disparity between

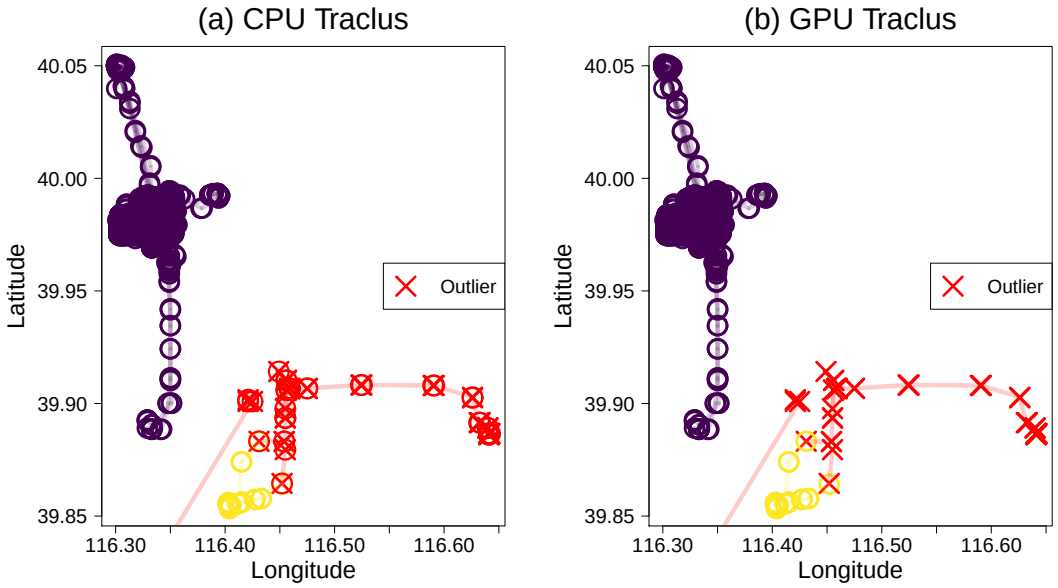

Figure 4.5: Classification of Geolife data line segments.

the two increased as well.

Table 4.3 shows the results of TRACLUS and GTRACLUS executed on Porto Taxi dataset. Initially, for fewer number of trajectories, the CPU Partition and the CPU Processing stages performed better than their GPU counterparts, performing as fast as 8.34X faster in partitioning stage and 1.71X faster in grouping stage for 100 trajectories.

The GPU grouping stage however, started gaining advantage over CPU grouping on 1000 trajectories, performing 3.43X better. This was observed as an increasing trend since for 100,000 trajectories, the GPU had gained 18.5X speedup over the CPU for clustering performance. The reason for this is the GPU's parallel architecture is well supported for independent distance calculations that G-DBSCAN utilizes. Every thread measures the parallel, perpendicular and angular distance independently and using hundreds of threads and overcomes the overheads produced by memory copy between main memory and device memory.

Epsilon	MinLines	NumTraj	CPU Partition (ms)	CPU Grouping (ms)	Number of CPU Clusters	GPU Partition (ms)	GPU Grouping (ms)	Number of GPU Clusters
0.001	5	100	13.25222	2.636242	0	110.6468	4.53263	0
0.001	10	1000	199.8734	254.3288	7	438.2512	73.9798	7
0.001	50	10000	2072.81	23912.72	2	977.5598	1270.718	2
0.001	100	100000	19887.82	2343540	0	8084.29	126669	0

Table 4.3: Benchmark Results on the Porto Dataset

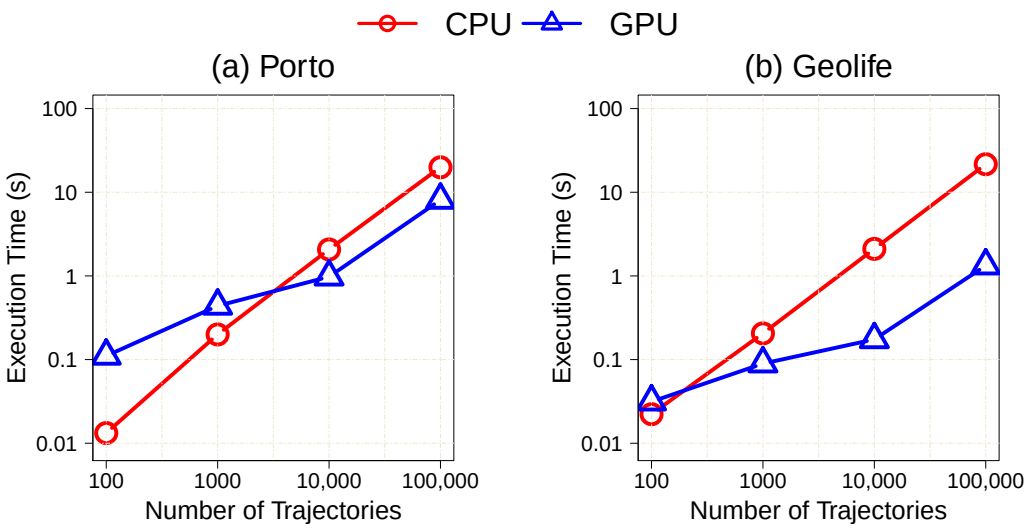

Figure 4.6: Partitioning performance on both datasets with *epsilon* set to 0.001.

The partitioning stage on the GPU for 1,000 trajectories was still taking twice as much time as on the CPU. At 10,000 trajectories it gained an advantage of 2.12X which increased to 2.46X at 100,000 trajectories. As mentioned earlier, the partitioning stage takes up less than 1% of the total time so a speedup or a slowdown of twice as much, as was obeserved, isn't as significant. If we analyze the cause of this behavior and the reason for GPU not providing significant speedups, we find out that partitioning trajectories is a semi-sequential process. It requires the algorithm to calculate MDL of every single point on a trajectory based on all the points it has seen so far. GPU kernels do not perform at their best when provided with sequential tasks. However, we managed to parallelize this process over multiple trajectories but because of a relatively quicker computational time of this stage the memory overhead for transferring data to GPU was not compensated by faster processing and hence the delays until the number of trajectories was large enough to overcome the transfer overhead. Figure 4.6(a) and 4.7(a) plot the results of Table 4.3 on a log scale on number of trajectories and execution time.

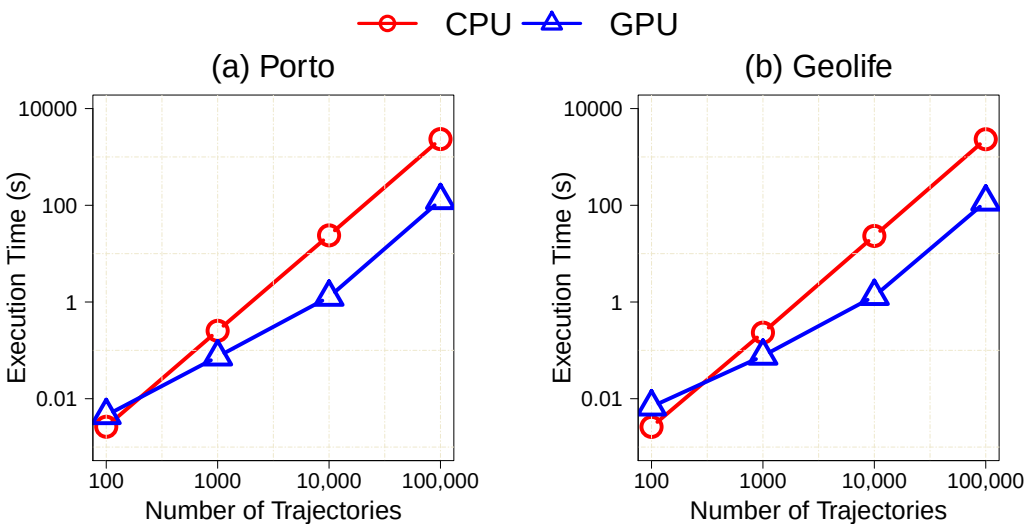

Figure 4.7: Grouping performance on both datasets with *epsilon* set to 0.001.

The experiment was conducted on Geolife dataset as well and the results are plotted in Figure 4.6(b) and 4.7(b). For the processing time on Geolife, the speedup obtained was similar, with the processing time on 100,000 trajectories on TRACLUS being 2,327.5 seconds compared to 118.2 seconds on GTRACLUS, a 19.69X speedup. However, as shown in Figure 4.6(b) GTRACLUS performed much better in the partitioning stage too this time. At 1,000 trajectories it was partitioning them 2.3X faster than TRACLUS going up to 16.3X on 100,000 trajectories.

4.5 Discussion

To summarize, this chapter explains the following:

- A novel GPU-based trajectory clustering algorithm, GTRACLUS, that employs the strengths of G-DBSCAN [3] and TRACLUS [37] to achieve trajectory clustering with high efficiency.
- The parallel architecture and a high number of threads available to us with the

GPU architecture, coupled with proposed parallel techniques that utilize this co-processor, resulted in 20X faster trajectory clustering than TRACLUS while providing the same results on multiple real-world datasets [68] [44].

5 Conclusions & Future Work

In this chapter, we present our findings and conclusions for our study on trajectory similarity queries, our comparison of GPU-based DBSCAN algorithms, and trajectory clustering on GPUs. We also discuss future work in these areas.

5.1 Conclusions

5.1.1 FastTopK: Trajectory Similarity Queries with GPUs

In Chapter 2 we proposed a novel technique for trajectory similarity queries using GPUs. The following is the summary of our discussion on trajectory similarity query processing algorithms:

- We introduced FastTopK [46], a novel and scalable top-K trajectory similarity query processing algorithm that efficiently uses the general-purpose programming capabilities of the GPU.

- This algorithm uses a new strategy to prune away trajectories that do not form part of the query result set, achieving up to 99.44% pruning, while using parallelism, load balancing, memory coalescing and GPU occupancy to perform what could have been to 6.56×10^{18} Euclidean distance comparisons with exhaustive search, in merely 158.69 seconds.

- This algorithm was compared against the current fastest top-K trajectory simi-

larity query processing algorithm, TKSimGPU, which already runs 3.8X faster than a multicore CPU implementation using up to 16 cores. Our proposed algorithm proved to be 3.36X faster than TKSimGPU on average, with speedups going up to 4.7X.

5.1.2 DBSCAN Clustering: Contemporary GPU-based Algorithm Analysis

In this work we also provided the first experimental comparison of the existing GPU-based DBSCAN clustering algorithms [45]. Chapter 3 provided a comparative analysis on contemporary GPU-based DBSCAN techniques. The summary of our discussion on this comparison study is as follows:

- We compared the following GPU-based clustering algorithms: Thapa et al.'s algorithm [59], CUDA D-Clust [6], and G-DBSCAN [3], and compared them against the R-tree-indexed single-threaded CPU DBSCAN [13] and multi-threaded CPU DBSCAN using three real-world datasets.

- Our experiments showed that G-DBSCAN, the GPU algorithm that scans clusters by using a breadth-first search through an adjacency list, is consistently the fastest across all datasets, up to 18X faster than the closest technique in terms of speed. But is also the least memory efficient. If a small dataset is to be used, this technique is sure to be the fastest compared to all the other techniques.

- CUDA-DClust, on the other hand, strikes a good balance between speed and memory efficiency: this algorithm is almost as fast as G-DBSCAN, while consuming orders of magnitude less RAM, which makes it a good choice for larger datasets. The memory consumption of CUDA-DClust was up to 166X less than

G-DBSCAN for 32k points in the Porto Taxi dataset in our experiments.

- Our experiments also showed that Thapa et al.'s algorithm, the GPU algorithm for DBSCAN that requires the least implementation effort, can be faster than the multi-threaded CPU DBSCAN algorithm, but only for larger datasets. For smaller datasets, e.g. datasets with fewer than 65,536 points, the user is better off saving on an extra co-processor and running the multi-threaded CPU DBSCAN algorithm.

- We also observed that the original single-threaded CPU DBSCAN algorithm with R-tree indexing shows execution times that are orders of magnitude slower than those observed in the GPU algorithms. Moreover, it gets worse when the number of points to be clustered increases, providing a great incentive to use techniques such as G-DBSCAN and CUDA-DClust instead.

5.1.3 GTRACLUS: A GPU-based Trajectory Clustering Algorithm

Following the results of our experiment comparing all the current GPU-based DBSCAN clustering algorithms we proposed a new, GPU-based technique for speeding up trajectory clustering. Chapter 4 details the new algorithm. The summary of our efforts in this area is as follows:

- TRACLUS [37], being the only algorithm that clusters trajectories as well as finds common subtrajectories is slow due to trajectory clustering being a $O(n^2)$ operation where n is the total number of points in all the trajectories. On top of that TRACLUS has two stages involving three distance measure calculations of every trajectory point.

- G-DBSCAN [3], the fastest GPU-based DBSCAN algorithm provided us with an opportunity to modify it to create a graph of line segments (instead of points) and perform breadth-first search on GPU between them to find clusters.

- Moreover, the Minimum Distance Length (MDL) calculation for partitioning stage was carried out using a parallel algorithm as well.

- The result was a new and extremely efficient, faster GPU-based trajectory clustering algorithm, named GTRACLUS. In our experiments, GTRACLUS provides the same results as TRACLUS but 20X faster, bringing down partitioning and clustering time from 40 minutes on CPU, to 2.2 minutes.

5.2 Future Work

Here we present some directions to take in order to build up on our findings and contributions.

5.2.1 FastTopK: Trajectory Similarity Queries with GPUs

For future work in trajectory similarity query processing, we intend to extend the proposed algorithm, FastTopK, to make it into an anytime top-K trajectory similarity query processing algorithm. Anytime algorithms [5] or *interruptible algorithms*, trade quality of the results with execution time. The result is a robust technique that can be utilized when it is not feasible to obtain the best quality results due to computational insufficiency. Anytime algorithms provide quick but inaccurate results to the user and increase the accuracy of the results over time. We also intend to add stream processing capabilities to this algorithm so that it can find top-K results in a database larger than the maximum available memory in GPUs, with trajectory data of potentially

millions of users over thousands of kilometers. FastTopK can also be extended by employing other trajectory similarity measures [43] such as ERP [8] , EDR [9], DTW [32], or the direction dependent distance measure proposed in [56]. These distance measures can be parallelized and incorporated into FastTopK, and can potentially provide useful insights by utilizing their strengths.

5.2.2 DBSCAN Clustering: Contemporary GPU-based Algorithm Analysis

To extend the comparative analysis of GPU-based DBSCAN algorithms, multiple devices can be employed to make the results more comprehensive. In our experiments, we executed all the algorithms on a single device, but having evaluation data from more devices could potentially generate useful insights into the behavior of the algorithms tested. Moreover, the comparative analysis of GPU-based DBSCAN algorithms will need updating if new GPU-based DBSCAN algorithms are proposed in the future.

5.2.3 GTRACLUS: A GPU-based Trajectory Clustering Algorithm

For future work in trajectory clustering, we would like to research applications of GPU-based parallel processing on other types of clustering algorithms for trajectories. Fuzzy cluster membership of trajectories is one such area [52]. In real-world applications, a trajectory can potentially be identified as a member of multiple clusters with varying percentage of membership. To the best of our knowledge, no work as been done in utilizing GPUs to make this type of clustering more efficient. Moreover, we intend to research distance measures other than the ones used in this thesis, all

of which do not take into account the direction of trajectories. Direction dependent distance measures can provide more insights into cluster membership of trajectories, and can be useful in real-world applications that are dependent on path, shape and direction of trajectories.

References

[1] In: *IEEE Trans. Parallel Distrib. Syst.* 27.9 (2016). ISSN: 1045-9219 (cit. on p. 23).

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications". In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. SIGMOD '98. Seattle, Washington, USA: ACM, 1998, pp. 94–105. ISBN: 0-89791-995-5. DOI: 10.1145/276304.276314. URL: http://doi.acm.org/10.1145/276304.276314 (cit. on p. 48).

[3] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha. "G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering". In: *Procedia Computer Science* 18 (2013). 2013 International Conference on Computational Science, pp. 369–378. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2013.05.200. URL: http://www.sciencedirect.com/science/article/pii/S1877050913003438 (cit. on pp. 18, 49, 58, 74, 90, 99, 102, 104).

[4] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. "OPTICS: Ordering Points to Identify the Clustering Structure". In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD

'99. Philadelphia, Pennsylvania, USA: ACM, 1999, pp. 49–60. ISBN: 1-58113-084-8. DOI: 10.1145/304182.304187. URL: http://doi.acm.org/10.1145/304182.304187 (cit. on p. 48).

[5] M. Boddy and T. L. Dean. "Deliberation scheduling for problem solving in time-constrained environments". In: *Artificial Intelligence* 67.2 (1994), pp. 245–285. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(94)90054-X. URL: http://www.sciencedirect.com/science/article/pii/000437029490054X (cit. on p. 104).

[6] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther. "Density-based Clustering Using Graphics Processors". In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. CIKM '09. Hong Kong, China: ACM, 2009, pp. 661–670. ISBN: 978-1-60558-512-3. DOI: 10.1145/1645953.1646038. URL: http://doi.acm.org/10.1145/1645953.1646038 (cit. on pp. 18, 49, 53, 58, 74, 102).

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. "A performance study of general-purpose applications on graphics processors using CUDA". In: *Journal of Parallel and Distributed Computing* 68.10 (2008). General-Purpose Processing using Graphics Processing Units, pp. 1370–1380. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2008.05.014. URL: http://www.sciencedirect.com/science/article/pii/S0743731508000932 (cit. on pp. 5, 14).

[8] L. Chen and R. Ng. "On the Marriage of Lp-norms and Edit Distance". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 792–

803. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316758 (cit. on pp. 22, 105).

[9] L. Chen, M. T. Özsu, and V. Oria. "Robust and Fast Similarity Search for Moving Object Trajectories". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 491–502. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066213. URL: http://doi.acm.org/10.1145/1066157.1066213 (cit. on pp. 22, 105).

[10] B. Dai and I. Lin. "Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition". In: *2012 IEEE Fifth International Conference on Cloud Computing*. June 2012, pp. 59–66. DOI: 10.1109/CLOUD.2012.42 (cit. on p. 48).

[11] H. Ding, G. Trajcevski, and P. Scheuermann. "Efficient Similarity Join of Large Sets of Moving Object Trajectories". In: *2008 15th International Symposium on Temporal Representation and Reasoning*. June 2008, pp. 79–87. DOI: 10.1109/TIME.2008.25 (cit. on p. 22).

[12] N. Docs. *Cuda C Best Practices Guide*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. [Online; accessed 14-Oct-2019]. 2018 (cit. on pp. 8, 10).

[13] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. "A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231. URL: http://dl.acm.org/citation.cfm?id=3001460.3001507 (cit. on pp. 16, 18, 19, 48, 49, 51, 74, 76, 86, 102).

[14] S. Gaffney, A. Robertson, P. Smyth, S. Camargo, and M. Ghil. "Probabilistic clustering of extratropical cyclones using regression mixture models". In: *Climate Dynamics* 29 (Sept. 2007), pp. 423–440. DOI: 10.1007/s00382-007-0235-z (cit. on pp. 78, 80).

[15] S. Gaffney and P. Smyth. "Trajectory Clustering with Mixtures of Regression Models". In: *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '99. San Diego, California, USA: ACM, 1999, pp. 63–72. ISBN: 1-58113-143-7. DOI: 10.1145/312129.312198. URL: http://doi.acm.org/10.1145/312129.312198 (cit. on pp. 77, 80).

[16] J. Gan and Y. Tao. "DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 519–530. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2737792. URL: http://doi.acm.org/10.1145/2723372.2737792 (cit. on p. 48).

[17] A. Ghose. *Tap: Unlocking the Mobile Economy*. The MIT Press, 2017. ISBN: 0262036274, 9780262036276 (cit. on p. 77).

[18] M. Götz, C. Bodenstein, and M. Riedel. "HPDBSCAN: Highly Parallel DBSCAN". In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. MLHPC '15. Austin, Texas: ACM, 2015, 2:1–2:10. ISBN: 978-1-4503-4006-9. DOI: 10.1145/2834892.2834894. URL: http://doi.acm.org/10.1145/2834892.2834894 (cit. on p. 48).

[19] M. Gowanlock and H. Casanova. "Distance threshold similarity searches on spatiotemporal trajectories using GPGPU". In: *2014 21st International Con-*

ference on High Performance Computing (HiPC). Dec. 2014, pp. 1–10. DOI: 10.1109/HiPC.2014.7116913 (cit. on p. 23).

[20] A. Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. Boston, Massachusetts: ACM, 1984, pp. 47–57. ISBN: 0-89791-128-8. DOI: 10.1145/602259.602266. URL: http://doi.acm.org/10.1145/602259.602266 (cit. on pp. 18, 48, 52).

[21] M. Hahsler, M. Piekenbrock, and D. Doran. "dbscan: Fast Density-Based Clustering with R". In: *Journal of Statistical Software* 91.1 (2019), pp. 1–30. DOI: 10.18637/jss.v091.i01 (cit. on p. 65).

[22] J. Han, M. Kamber, and J. Pei. *Data mining concepts and techniques, third edition*. 2012. URL: http://www.amazon.de/Data-Mining-Concepts-Techniques-Management/dp/0123814790/ref=tmm_hrd_title_0?ie=UTF8&qid=1366039033&sr=1-1 (cit. on pp. 16, 47, 48).

[23] P. Harish and P. J. Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA". In: *Proceedings of the 14th International Conference on High Performance Computing*. HiPC'07. Goa, India: Springer-Verlag, 2007, pp. 197–208. ISBN: 3-540-77219-7, 978-3-540-77219-4. URL: http://dl.acm.org/citation.cfm?id=1782174.1782200 (cit. on p. 59).

[24] M. Harris. *Optimizing Parallel Reduction in CUDA*. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. [Online; accessed 14-Oct-2019]. 2016 (cit. on p. 30).

[25] N. J. Hill, I. Hussein, K. Davis, E. Ma, T. Spivey, A. Ramey, W. Puryear, S. Das, R. Halpin, X. Lin, N. Fedorova, D. Suarez, W. Boyce, and J. Runstadler. "Reassortment of Influenza A Viruses in Wild Birds in Alaska before H5 Clade

2.3.4.4 Outbreaks". In: *Emerging Infectious Diseases* 23 (Apr. 2017), pp. 654–657. DOI: 10.3201/eid2304.161668 (cit. on p. 77).

[26] J. Horne, E. Garton, S. Krone, and J. Lewis. "Analyzing animal movements using Brownian bridges". English (US). In: *Ecology* 88.9 (Sept. 2007), pp. 2354–2363. ISSN: 0012-9658. DOI: 10.1890/06-0957.1 (cit. on pp. 2, 12).

[27] S. Huang, S. Xiao, and W. Feng. "On the energy efficiency of graphics processing units for scientific computing". In: *2009 IEEE International Symposium on Parallel Distributed Processing*. May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160980 (cit. on p. 18).

[28] J. Jang and H. Jiang. "DBSCAN++: Towards fast and scalable density clustering". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, Sept. 2019, pp. 3019–3029. URL: http://proceedings.mlr.press/v97/jang19a.html (cit. on pp. 18, 48).

[29] L. Kaufman and P. J. Rousseeuw. *Clustering by means of medoids*. Ed. by I. D. Y and editor. Amsterdam: 1987 (cit. on p. 48).

[30] M. Kaul, B. Yang, and C. S. Jensen. "Building Accurate 3D Spatial Networks to Enable Next Generation Intelligent Transportation Systems". In: *2013 IEEE 14th International Conference on Mobile Data Management*. Vol. 1. June 2013, pp. 137–146. DOI: 10.1109/MDM.2013.24 (cit. on p. 63).

[31] R. E. Kenward, S. S. Walls, and K. H. Hodder. "Life path analysis: scaling indicates priming effects of social and habitat factors on dispersal distances". In: *Journal of Animal Ecology* 70.1 (2001), pp. 1–13. DOI: 10.1111/j.1365-2656.2001.00464.x. eprint: https://besjournals.onlinelibrary.wiley.

com/doi/pdf/10.1111/j.1365-2656.2001.00464.x. URL: https://besjournals.onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2656.2001.00464.x (cit. on pp. 2, 12).

[32] E. Keogh and C. A. Ratanamahatana. "Exact Indexing of Dynamic Time Warping". In: *Knowl. Inf. Syst.* 7.3 (Mar. 2005), pp. 358–386. ISSN: 0219-1377. DOI: 10.1007/s10115-004-0154-9. URL: http://dx.doi.org/10.1007/s10115-004-0154-9 (cit. on p. 105).

[33] N. Labs. *CUDA CUB*. https://nvlabs.github.io/cub/. [Online; accessed 14-Oct-2019]. 2018 (cit. on pp. 32, 93).

[34] E. Leal and L. Gruenwald. "DynMDL: A Parallel Trajectory Segmentation Algorithm". In: *2018 IEEE International Congress on Big Data (BigData Congress)*. July 2018, pp. 215–218. DOI: 10.1109/BigDataCongress.2018.00036 (cit. on p. 86).

[35] E. Leal, L. Gruenwald, J. Zhang, and S. You. "TKSimGPU: A parallel top-K trajectory similarity query processing algorithm for GPGPUs". In: *2015 IEEE International Conference on Big Data (Big Data)*. Oct. 2015, pp. 461–469. DOI: 10.1109/BigData.2015.7363787 (cit. on pp. 14, 20, 23, 31, 33, 46).

[36] E. Leal, L. Gruenwald, J. Zhang, and S. You. "Towards an Efficient Top-K Trajectory Similarity Query Processing Algorithm for Big Trajectory Data on GPGPUs". In: *2016 IEEE International Congress on Big Data (BigData Congress)*. June 2016, pp. 206–213. DOI: 10.1109/BigDataCongress.2016.33 (cit. on pp. 14, 23).

[37] J.-G. Lee, J. Han, and K.-Y. Whang. "Trajectory Clustering: A Partition-and-group Framework". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007,

pp. 593–604. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247546. URL: http://doi.acm.org/10.1145/1247480.1247546 (cit. on pp. 17, 76, 78, 80, 81, 99, 103).

[38] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 451–460. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1816021. URL: http://doi.acm.org/10.1145/1815961.1816021 (cit. on pp. 4, 18).

[39] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-Y. Ma. "Mining User Similarity Based on Location History". In: *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS '08. Irvine, California: ACM, 2008, 34:1–34:10. ISBN: 978-1-60558-323-5. DOI: 10.1145/1463434.1463477. URL: http://doi.acm.org/10.1145/1463434.1463477 (cit. on p. 77).

[40] Z. Li, J.-G. Lee, X. Li, and J. Han. "Incremental Clustering for Trajectories". In: *Proceedings of the 15th International Conference on Database Systems for Advanced Applications - Volume Part II*. DASFAA'10. Tsukuba, Japan: Springer-Verlag, 2010, pp. 32–46. ISBN: 3-642-12097-0, 978-3-642-12097-8. DOI: 10.1007/978-3-642-12098-5_3. URL: http://dx.doi.org/10.1007/978-3-642-12098-5_3 (cit. on pp. 78, 80).

[41] S. P. Lloyd. "Least squares quantization in pcm". In: *IEEE Transactions on Information Theory* 28 (1982), pp. 129–137 (cit. on p. 48).

[42] C. Ma, H. Lu, L. Shou, and G. Chen. "KSQ: Top-k Similarity Query on Uncertain Trajectories". In: *IEEE Transactions on Knowledge and Data Engineering* 25.9 (Sept. 2013), pp. 2049–2062. ISSN: 1041-4347. DOI: 10.1109/TKDE.2012.152 (cit. on p. 22).

[43] N. Magdy, M. A. Sakr, T. Mostafa, and K. El-Bahnasy. "Review on trajectory similarity measures". In: *2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS)*. Dec. 2015, pp. 613–619. DOI: 10.1109/IntelCIS.2015.7397286 (cit. on p. 105).

[44] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. "Predicting Taxi–Passenger Demand Using Streaming Data". In: *IEEE Transactions on Intelligent Transportation Systems* 14.3 (Sept. 2013), pp. 1393–1402. ISSN: 1524-9050. DOI: 10.1109/TITS.2013.2262376 (cit. on pp. 4, 14, 31, 63, 93, 100).

[45] H. Mustafa, E. Leal, and L. Gruenwald. "An Experimental Comparison of GPU Techniques for DBSCAN Clustering". In: *2019 IEEE International Workshop on Benchmarking, Performance Tuning and Optimization for Big Data Applications*. Dec. 2019 (cit. on pp. 20, 102).

[46] H. Mustafa, E. Leal, and L. Gruenwald. "FastTopK: A Fast Top-K Trajectory Similarity Query Processing Algorithm for GPUs". In: *2018 IEEE International Conference on Big Data (Big Data)*. Dec. 2018, pp. 542–547. DOI: 10.1109/BigData.2018.8621907 (cit. on pp. 20, 22, 24, 31, 33, 45, 101).

[47] S. Nutanong, E. H. Jacox, and H. Samet. "An Incremental Hausdorff Distance Calculation Algorithm". In: *Proc. VLDB Endow.* 4.8 (May 2011), pp. 506–517. ISSN: 2150-8097. DOI: 10.14778/2002974.2002978. URL: http://dx.doi.org/10.14778/2002974.2002978 (cit. on pp. 12, 22).

[48] Nvidia. *CUDA*. https://docs.nvidia.com/cuda/. [Online; accessed 14-Oct-2019]. 2019 (cit. on pp. 5–7, 64).

[49] Nvidia. *Nvidia Turing GPU Architecture*. 2019 (cit. on p. 6).

[50] E. P. *Taxi Service Trajectory Prediction Challenge*. http://www.geolink.pt/ecmlpkdd2015-challenge/whoweare.html. [Online; accessed 14-Oct-2019]. 2018 (cit. on p. 31).

[51] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. Choudhary. "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2012, pp. 1–11. DOI: 10.1109/SC.2012.9 (cit. on p. 48).

[52] N. Pelekis, I. Kopanakis, E. E. Kotsifakos, E. Frentzos, and Y. Theodoridis. "Clustering Uncertain Trajectories". In: *Knowl. Inf. Syst.* 28.1 (July 2011), pp. 117–147. ISSN: 0219-1377. DOI: 10.1007/s10115-010-0316-x. URL: http://dx.doi.org/10.1007/s10115-010-0316-x (cit. on pp. 79, 80, 105).

[53] S. Ranu, Deepak P, A. D. Telang, P. Deshpande, and S. Raghavan. "Indexing and matching trajectories under inconsistent sampling rates". In: *2015 IEEE 31st International Conference on Data Engineering*. Apr. 2015, pp. 999–1010. DOI: 10.1109/ICDE.2015.7113351 (cit. on p. 22).

[54] J. H. Reif. *Synthesis of Parallel Algorithms*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 155860135X (cit. on p. 10).

[55] J. Rissanen. "Modeling by shortest data description". In: *Automatica* 14.5 (1978), pp. 465–471. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-

1098(78)90005-5. URL: http://www.sciencedirect.com/science/article/pii/0005109878900055 (cit. on p. 84).

[56] G.-P. Roh and S.-w. Hwang. "NNCluster: An Efficient Clustering Algorithm for Road Network Trajectories". In: *Proceedings of the 15th International Conference on Database Systems for Advanced Applications - Volume Part II*. DASFAA'10. Tsukuba, Japan: Springer-Verlag, 2010, pp. 47–61. ISBN: 3-642-12097-0, 978-3-642-12097-8. DOI: 10.1007/978-3-642-12098-5_4. URL: http://dx.doi.org/10.1007/978-3-642-12098-5_4 (cit. on pp. 79, 80, 105).

[57] D. Rohr, S. Kalcher, M. Bach, A. A. Alaqeeliy, H. M. Alzaidy, D. Eschweiler, V. Lindenstruth, S. B. Alkhereyfy, A. Alharthiy, A. Almubaraky, I. Alqwaizy, and R. B. Suliman. "An Energy-Efficient Multi-GPU Supercomputer". In: *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*. Aug. 2014, pp. 42–45. DOI: 10.1109/HPCC.2014.14 (cit. on p. 3).

[58] U. D. o. T. (USDOT). *Next Generation Simulation (NGSIM) Vehicle Trajectories and Supporting Data*. https://catalog.data.gov/dataset/next-generation-simulation-ngsim-vehicle-trajectories. [Online; accessed 14-Oct-2019]. 2018 (cit. on p. 63).

[59] R. J. Thapa, C. Trefftz, and G. Wolffe. "Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases". In: *2010 IEEE International Conference on Electro/Information Technology*. May 2010, pp. 1–5. DOI: 10.1109/EIT.2010.5612134 (cit. on pp. 18, 49, 51–53, 74, 102).

[60] W. Wang, J. Yang, and R. R. Muntz. "STING: A Statistical Information Grid Approach to Spatial Data Mining". In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 186–195. ISBN: 1-55860-470-7. URL: http://dl.acm.org/citation.cfm?id=645923.758369 (cit. on p. 48).

[61] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013. ISBN: 9780321809469. URL: https://books.google.com/books?id=KUxsAQAAQBAJ (cit. on p. 10).

[62] B. Zhang, Y. Shen, Y. Zhu, and J. Yu. "A GPU-Accelerated Framework for Processing Trajectory Queries". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. Apr. 2018, pp. 1037–1048. DOI: 10.1109/ICDE.2018.00097 (cit. on p. 4).

[63] J. Zhang, S. You, and L. Gruenwald. "U2STRA: High-performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs". In: *Proceedings of the 2012 ACM Workshop on City Data Management Workshop*. CDMW '12. Maui, Hawaii, USA: ACM, 2012, pp. 5–12. ISBN: 978-1-4503-1709-2. DOI: 10.1145/2390226.2390229. URL: http://doi.acm.org/10.1145/2390226.2390229 (cit. on p. 23).

[64] Y. Zheng. "Location-Based Social Networks: Users". In: *Computing with Spatial Trajectories*. Ed. by Y. Zheng and X. Zhou. New York, NY: Springer New York, 2011, pp. 243–276. ISBN: 978-1-4614-1629-6. DOI: 10.1007/978-1-4614-1629-6_8. URL: https://doi.org/10.1007/978-1-4614-1629-6_8 (cit. on pp. 2, 77).

[65] Y. Zheng. "Trajectory Data Mining: An Overview". In: *ACM Transaction on Intelligent Systems and Technology* (Sept. 2015). URL: https://www.microsoft.

com/en-us/research/publication/trajectory-data-mining-an-overview/ (cit. on p. 77).

[66] Y. Zheng. "Trajectory Data Mining: An Overview". In: *ACM Trans. Intell. Syst. Technol.* 6.3 (May 2015), 29:1–29:41. ISSN: 2157-6904. DOI: 10.1145/2743025. URL: http://doi.acm.org/10.1145/2743025 (cit. on p. 31).

[67] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. "Urban Computing: Concepts, Methodologies, and Applications". In: *ACM Transaction on Intelligent Systems and Technology* (Oct. 2014). URL: https://www.microsoft.com/en-us/research/publication/urban-computing-concepts-methodologies-and-applications/ (cit. on pp. 2, 77).

[68] Y. Zheng, X. Xie, and W.-Y. Ma. "GeoLife: A Collaborative Social Networking Service among User, location and trajectory". In: *IEEE Data(base) Engineering Bulletin* (June 2010). URL: https://www.microsoft.com/en-us/research/publication/geolife-a-collaborative-social-networking-service-among-user-location-and-trajectory/ (cit. on pp. 2, 3, 14, 31, 93, 100).

[69] Y. Zheng, X. Xie, and W.-Y. Ma. "Mining Interesting Locations and Travel Sequences From GPS Trajectories". In: *Proceedings of International conference on World Wide Web 2009*. WWW 2009. Apr. 2009. URL: https://www.microsoft.com/en-us/research/publication/mining-interesting-locations-and-travel-sequences-from-gps-trajectories/ (cit. on p. 77).