GTraclus: A Local Trajectory Clustering Algorithm for GPUs

Hamza Mustafa
Microsoft
Bellevue, Washington, USA
musta067@d.umn.edu

Clark Barrus
School of Computer Science
University of Oklahoma
Norman, OK, USA
clark.barrus@ou.edu

Eleazar Leal
Department of Computer Science
University of Minnesota Duluth
Duluth, MN, USA
eleal@d.umn.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

***Abstract*—Due to the high availability of location-based sensors like GPS, it has been possible to collect large amounts of spatio-temporal data in the form of trajectories, each of which is a sequence of spatial locations that a moving object occupies in space as time progresses. Many applications, such as intelligent transportation systems and urban planning, can benefit from clustering the trajectories of cars in each locality of a city in order to learn about traffic behavior in each neighborhood. However, the immense and ever-increasing volume of trajectory data and the concept drift present in city traffic constitute scalability challenges that have not been addressed. In order to fill this gap, we propose the first GPU algorithm for local trajectory clustering, called GTraclus. We present a parallelized trajectory partitioning algorithm which simplifies trajectories into line segments using the Minimum Description Length (MDL) principle. We evaluated our proposed algorithm using two large real-life trajectory datasets and compared it against a multicore CPU version, which we call MC-Traclus, of the popular trajectory clustering algorithm, Traclus; our experiments showed that GTraclus had on average up to 24X faster execution time when compared against MC-Traclus.**

Keywords—trajectory clustering, GPU, spatio-temporal data

I. Introduction

Easily attainable GPS technology and cheap storage space have led to an unprecedented amount of trajectory data, where a trajectory is the time-ordered sequence of positions, i.e., latitude and longitude, that a moving object occupies in space as time passes. This provides a great opportunity for analysis of similar patterns on time-varying data by clustering the trajectories into groups containing similar trajectories. Such analysis has a broad range of applications in bird migration pattern identification, location-based social networks [1], recommendation of travel locations of interest based on common trajectories [2], finding users with similar life experiences based on their trajectories [3], intelligent transportation systems, and urban computing [4]. Trajectory clustering 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 wireless network, can send personalized advertising information to customers based on their paths inside the mall [5].

An important consideration for clustering trajectories is whether the elements that are going to be clustered are entire trajectories, in which case we say that we perform *global* trajectory clustering, or whether the elements to be clustered are not entire trajectories but sub-trajectories, which gives rise to the problem of *local* trajectory clustering. In many applications, clustering entire trajectories may not provide important insights into the common shorter paths the objects take, as real-world objects do not always take similar paths for the entirety of their journeys. For example, when predicting hurricane landfall based on hurricane trajectories, meteorologists are more interested in clustering hurricane behaviors near the coastline or at the sea rather than on the entire hurricane trajectories [6]. Similarly, when examining the effects of vehicular traffic on animal movement, distribution, and habitat use, zoologists are more interested in common behaviors of animal trajectories near roads [7]. These problems can be solved with Traclus, a well-known local trajectory clustering algorithm for single-core CPUs [8].

Despite its wide range of applications, Traclus does not scale with large trajectory datasets. This problem along with the very large and ever-increasing sizes of spatio-temporal datasets and with the presence of concept drift in the previously mentioned applications, gives rise to a need for parallel local trajectory clustering algorithms. One way to address this problem is to utilize Graphics Processing Units (GPUs), which are parallel processors that can provide efficient and massively parallel computation with high instruction throughput and memory bandwidth, compared even to multicore CPUs [9]. However, developing algorithms for GPUs is not without challenges, as the latter have several idiosyncrasies that need to be addressed in order to attain the high performance throughput for which GPUs are known [10]. Among these idiosyncrasies are the small memory space of GPUs and that the interfaces through which they are connected to the computer (e.g., the PCIe bus) have low throughput when compared to their instruction throughput.

Despite the advantages of GPUs, no algorithm exists that exploits this architecture for *local* trajectory clustering. To address this gap, we introduce GTraclus, a novel GPU algorithm for local trajectory clustering. GTraclus includes a novel trajectory partitioning algorithm for GPUs that uses the Minimum Description Length principle (MDL) and a novel GPU algorithm for trajectory segment clustering. We analyze the performance of GTraclus when applied to two large, real-world datasets, GeoLife [2] and Porto [11], and compare its performance with that of a multicore CPU version of Traclus, which we call MC-Traclus. The contributions of this paper are the following: 1) A novel GPU algorithm, named GTraclus, for trajectory partitioning according to the Minimum Description Length principle; 2) a GPU algorithm to cluster segments with a breadth-first search on a graph whose nodes are the partitioned line segments; and 3) a comprehensive set of experiments demonstrating the performance and scalability of GTraclus clustering hundreds of thousands of trajectories from real-world datasets.

The remainder of this paper is organized as follows: Section II presents background concepts and related work; Section III contains the description of the proposed GTraclus algorithm; Section IV contains a thorough performance analysis; and finally, Section V presents conclusions and future work.

II. Background and Related Work

In this section, we provide the background material necessary to follow the discussions on GPUs, local trajectory clustering, and present related work.

GPUs. GPUs are highly parallel processors connected to the main computer through an interface like PCIe and can achieve up to an order of magnitude of higher throughput than comparable multicore CPUs [9]. GPU programs are organized into *kernels* [12], which are C-like functions called from within the CPU, also called the *host*. Kernels launch a *grid* of thousands of simultaneously executing *threads*, which are grouped into *blocks*. The GPU's memory space is separate from the host's, which makes it necessary to send all input data through the PCIe bus before any processing can take place in the GPU, and to send all output data from the GPU back to the host. The memory space of GPUs is also hierarchical: threads can access their own individual local memory registers; threads in a block can cooperate by using the larger block-wide shared memory; and threads across different blocks all have access to the slower but bigger global GPU memory.

In order to use GPUs to exploit the parallelism present in many algorithms, it is necessary to address the research challenges of this architecture. Among these challenges are the following: 1) *global memory coalescing*, which consists in a reduction in the contention for the GPU's global memory that results from having consecutive threads access adjacent memory locations [13]; 2) *low throughput of the GPU-host interface*. Since GPUs are connected to the host through relatively low throughput interfaces such as PCIe, it is essential that communication through the GPU-host interface is minimized; and 3) *load balancing*. GPU kernels must make sure that different threads and blocks have an equal amount of work so that a single thread or block does not dominate the total execution time.

Trajectory clustering. The problem of *trajectory clustering*, also called *global trajectory clustering*, consists in that given a dataset of trajectories *D* and a similarity measure between any two trajectories *s*, find a collection of mutually disjoint subsets, also called clusters, of *D* such that the trajectories belonging to any cluster *c* are more similar to each other according to *s* than they are to trajectories in other clusters different from *c*. Due to the importance of trajectory clustering applications, there are several works devoted to the study of this problem [14] [15] [16].

However, there are applications where clustering the entire trajectories may not provide insights into the common shorter paths that the objects took because real-world objects do not always take similar paths *for the entirety* of their journeys, instead they take similar paths *for only a portion* of them. For example, when clustering the trajectories of vehicles moving in a large city like Beijing, most people do not have very similar trajectories because they live and work in different places. However, if the trajectories are first broken into sub-trajectories and then clustered, then it is possible to discover, for example, that many vehicles drive on a specific highway. Based on this observation, the problem of *local trajectory clustering* [8] was introduced, which consists in that given a dataset of trajectories *D* and a similarity measure between any two line segments *s*, find a collection of mutually disjoint sets, also called clusters, of sub-trajectories of trajectories in *D* such that the sub-trajectories belonging to any cluster *c* are more similar to each other according to *s* than they are to sub-trajectories in clusters different from *c*.

The Traclus algorithm [8] was proposed to solve the local trajectory clustering problem and it works in two phases: it first partitions trajectories into line segments, and then clusters the line segments. Traclus uses the Minimum Description Length (MDL) principle to approximate the best representation for a trajectory while losing as little information as possible. Other works devoted to trajectory clustering are TCMM [17], CenTra-I-FCM [18] and NNCluster [19], neither of which can perform *local* trajectory clustering.

Despite the many advantages of GPUs, e.g. their availability in almost all kinds of computing devices, none of the clustering algorithms has been developed to address the issues of GPUs. To the extent of our knowledge, the only other GPU trajectory clustering algorithms are G-Tra-POPTICS [20], a density-based point clustering algorithm for global trajectory clustering, not for local trajectory clustering like ours, and the work by Gudmundsson and Valladares [21], which, unlike GTraclus, finds clusters of similar sub-trajectories *within a single trajectory* and makes use of the Fréchet distance and not MDL.

III. Proposed Algorithm

In this section, we present our proposed algorithm, GTraclus, for local trajectory clustering on GPUs.

A. Overview

GTraclus is a GPU algorithm for local trajectory clustering that receives as inputs two numbers: *minLines* and *Epsilon*. This algorithm works in two stages executed in succession: a partitioning stage and a grouping stage. In its partitioning stage, GTraclus uses the Minimum Description Length (MDL) principle to partition trajectories into line segments, and in its grouping stage, it uses a GPU density-based clustering algorithm to cluster similar line segments. GTraclus also includes several optimization strategies for GPUs to bring the computation time down. We now provide a brief overview of each of these stages.

In its partitioning stage, GTraclus uses separate GPU threads to partition each input trajectory by identifying *characteristic points*, which are the points belonging to the trajectory that best partition it into line segments in terms of MDL cost [8]. To discover the characteristic points of trajectory, a GPU thread traverses each point of the latter while comparing the MDL cost of either including or not including the current point. If the current point under consideration leads to a greater overall MDL cost for the trajectory it belongs to, then the *previous* point is classified as a characteristic point.

The grouping phase of GTraclus performs density-based clustering of the trajectory segments on the GPU using a *reachability graph*. This graph has as vertex or node set the set of all segments produced in the partitioning stage, and its edge set is constructed by having an edge between any two vertices that lie within a segment distance [8] of *Epsilon*. Then, it is possible to identify the core segments, i.e., those nodes that have at least *minLines* many nodes within a segment distance of *Epsilon*, then the border segments, i.e., those nodes that are not core, but such that there is a path in the reachability graph from them to a core segment, and the noise segments, which are all other segments that are neither core nor border. Then, by doing successive BFS traversals on the reachability graph starting from different core segments p, it is possible to identify all the nodes reachable from p, which are the members of the cluster to which p belongs.

B. Partitioning Stage

The first stage of GTraclus partitions the input trajectory dataset D using the GPU-Partition function called in Line 1 of the function GTraclus in Algorithm 1. Given a trajectory $T = T_1, T_2, \ldots, T_{\text{size}(T)}$, partitioning T according to the MDL principle consists in finding a subsequence $\{T_{c_0}, T_{c_1}, \ldots, T_{c_{l-1}}\}$ of points of T, each of which is called a *characteristic point*, such that the MDL cost, defined as $L(H) + L(D|H)$ is minimized, where $L(H)$ and $L(D|H)$ are defined as:

$$L(H) = \sum_{j=1}^{l-2} \log_2 \left(Length\left(T_{c_j}T_{c_{j+1}}\right)\right)$$

$$L(D|H) = \sum_{j=1}^{l-2} \sum_{k=c_j}^{c_{j+1}-1} \log_2 \left(d_\perp \left(T_{c_j}T_{c_{j+1}}, T_kT_{k+1}\right)\right) + \log_2 \left(d_\theta \left(T_{c_j}T_{c_{j+1}}, T_kT_{k+1}\right)\right).$$

In the above definition, $T_{c_j}T_{c_{j+1}}$ denotes the segment from T_{c_j} to $T_{c_{j+1}}$ and $Length\left(T_{c_j}T_{c_{j+1}}\right)$ is the Euclidean distance between its endpoints, $d_\perp$ is the orthogonal distance [17] between two segments L_i and L_j where $\|L_i\| \geq \|L_j\|$, and it is defined as follows: $d_\perp(L_i, L_j) = (l_{\perp 1}^2 + l_{\perp 2}^2) \,/\, (l_{\perp 1} + l_{\perp 2})$, where $l_{\perp 1}$ is the distance from one of the endpoints of L_j to L_i, $l_{\perp 2}$ is the distance from the other endpoint of L_j to L_i. Also, d_θ is the angular distance between those two segments and is defined as:

$$d_\theta(L_i, L_j) = \begin{cases} \|L_j\| \cdot \sin(\theta), & if\ 0 \leq \theta \leq \pi/2 \\ \|L_j\|, & if\ \pi/2 \leq \theta \leq \pi. \end{cases}$$

To solve the MDL trajectory partitioning problem, we follow Traclus's approximate partitioning algorithm [8]. We parallelize the problem on the GPU by assigning different trajectories to different threads. The first step in GTraclus's partitioning stage is to calculate the number of segments for each trajectory, which is done by the CountPartitions kernel. In this kernel, each thread is in charge of sequentially traversing the points of its assigned trajectory seeking for characteristic points. The number of characteristic points determines the number of segments of a trajectory. This kernel, called in Line 2 of the Function GPU-Partition in Algorithm 1, returns the array *dSegs* containing the number of segments for each trajectory.

Once the number of segments for each trajectory is known, it is possible to actually allocate space in the GPU's global memory to hold the trajectory segments. To accomplish this, an inclusive scan is executed over the array *dSegs*, as shown in Line 4 of the Function GPU-Partition in Algorithm 1. This operation can be performed efficiently on GPUs and is used in this case to compute the offsets in the *segments* array in which each trajectory's segments will reside. The original dataset is then discarded to free GPU memory. As mentioned, the segments are computed and stored using Traclus's approximate partitioning algorithm in parallel, as shown in Lines 5–8 of GPU-Partition.

```
Function GTraclus (Trajectory Set D, Double epsilon, Double minLines)
Performs local trajectory clustering on D in the GPU using the values of Epsilon and minLines
1. segments ← GPU-Partition(D)
2. labels ← G-TrajScan(segments, epsilon, minLines)
3. return (segments, labels)
```

```
Function GPU-Partition (Trajectory Set D)
Each trajectory in D is partitioned in parallel according to the MDL principle
1. for each trajectory t in D do in parallel
2.     dSegs[t] ← CountPartitions(t)
3. done
4. dSegs ← InclusivePrefixSum(dSegs)
5. for each trajectory t in D do in parallel
6.     segments ← FillPartitions(t, dSegs)
7. done
8. return segments
```

Algorithm 1. Pseudo-code of the GTraclus Algorithm

By first counting the number of partitions, allocating space for the results on the host and then calculating and saving those partitions in GPU memory, the partitioning stage can take place entirely in GPU memory. If each thread allocated its own separate space for the segments of its trajectory, each partition would be in an independent location in memory and the host, and future kernels would have to deal with different data locations. The strategy of maintaining all of the segments in one array indexed by *dSegs* ensures that the partitions are aligned in one array and in contiguous memory for use in the following kernels.

C. Grouping Stage

In the grouping stage, called in Line 2 of the GTraclus function in Algorithm 1, the line segments produced by the partitioning stage are clustered. To do this, we present a new density-based segment clustering algorithm for GPUs. Unlike the existing G-DBSCAN [22], which is designed for density-based clustering of points, the algorithm proposed here clusters trajectory segments, which poses a different challenge concerning the arrangement of data in memory in order to guarantee global memory coalescing in the GPU.

In Line 1 of the procedure G-TrajScan in Algorithm 2, our proposed algorithm creates a density-connectedness graph whose vertices are the segments of all trajectories. There exists an edge between any two vertices in this graph if and only if their corresponding segments lie within an *Epsilon* distance of each other, measured according to the segment distance of Traclus [8]. Once this graph is created, GTraclus performs a sequence of parallel breadth-first searches (BFS) on the segments in order to find segment clusters; each separate BFS search gives rise to a different trajectory cluster. This strategy is highly efficient on GPUs because the large amount of distance calculations between trajectory segments can be parallelized. Given any vertex, all other vertices that are reachable from it, meaning that there is a path in the graph between them, are labeled as members of the same cluster. All the segments that are not part of any cluster are labeled as noise.

Since our proposed segment clustering algorithm clusters line segments, it stores for each segment the coordinates of its start and end points, the index of the trajectory to which it belongs, and the identifier of the cluster to which it will belong. To this end, Lines 1–4 of the Make-Graph function in Algorithm 2 count the number of segments within an *Epsilon* distance of each segment, then Line 5 computes a prefix sum over the number of neighbors of each segment with the purpose of allocating space for the adjacency matrix. Lines 6–11 then compute the adjacency matrix by checking in parallel which segments are within an *Epsilon* distance for each node.

Once the reachability graph is created, a GPU-based BFS is initiated in Line 2 of the G-TrajScan procedure in Algorithm 2. Each core segment (line segments with at least *minLines* many segments within a segment distance of *Epsilon*) will become the starting point of a BFS search and each thread goes into the adjacency lists of density-connected line segments and marks them as visited (Lines 1–8 in the Identify-Clusters procedure of Algorithm 2). All density-connected line segments [23] get the same cluster label, and the process is conducted in parallel. Once all segments are marked as visited, the algorithm concludes, and clustering data is provided as output. The GPU-BFS is designed as a sequence of BFS kernel calls where each thread is assigned a node on the frontier, which consists of the nodes in the adjacency list of the previous frontier, until the entire frontier is explored. This allows the GPU to explore the entire frontier of the breadth-first search at the same time.

Since each thread performs each one of the calculations in parallel, the threads can make full use of their local thread registers to store and sum the distance and vector calculations. These calculations are designed to contain no branching logic so different thread instructions will not diverge, allowing the GPU to achieve high performance. Since all of the partitioned line segments are stored in a single array and threads access this array in a serial fashion when loading the partitions to perform distance calculations, the GPU is able to coalesce the data transfers from global memory.

IV. Performance Analysis

In this section, we describe the datasets, hardware and the setup used in the experiments presented in this paper. We compare GTraclus against MC-Traclus, a multicore algorithm based on Traclus that we wrote for this purpose and that in these experiments we run with 24 threads. MC-Traclus is identical to Traclus, except it parallelizes the distance computation between segment pairs by assigning different threads to different pairs.

A. Datasets

For our experiments, we used two real datasets: Geolife [2] and the Taxi Service Trajectory Prediction Challenge dataset [11], which we will refer to as Porto. GeoLife contains the trajectories of people and cars as they move through the city of Beijing, while the Porto trajectories log taxis as they move through the city of Porto in Portugal. Both datasets contain a considerable amount of trajectory data, Geolife has 17,621, which were broken down into over 100,000 trajectories, and

Procedure G-TrajScan (Segment Set *D,* double *Epsilon,* double *minLines*)
Performs the grouping stage of GTraclus on *D* in the GPU using the values of *Epsilon* and *minLines*

```
 1. graph ← Make-Graph(D, Epsilon, minLines)
 2. labels ← Identify-Clusters(graph, Epsilon, minLines)
 3. return labels
```

Function Make-Graph (Segment Set *D,* double *Epsilon,* double *minLines*)
Constructs the reachability graph on the set of segments *D*

```
 1. for p in D do in parallel numNeighbors[p] ← 0 end for
 2. for each pair(p,q) of segments in D do in parallel
 3.    if dist(p,q) < Epsilon then numNeighbors[p]++ end if
 4. done
 5. startPos ← ExclusivePrefixSum(numNeighbors)
 6. for each pair(p,q) of segments in D do in parallel
 7.    if dist(p,q) < Epsilon then
 8.       Add q to the adjacency list of p and vice versa
 9.    end if
10. end for
11. return new Graph(adjacency, startPos)
```

Procedure Identify-Clusters (Graph *G,* double *Epsilon,* double *minLines*)
Performs a BFS on the graph *G* in order to discover clusters

```
 1. clusterID ← 0
 2. for each node v do visited[v] ← false done
 3. for each node v in G do
 4.    if not visited(v) and v is a core segment then
 5.       visited[v] ← true; labels[v] ← clusterID
 6.       GPU-BFS(v, G, Epsilon, minLines, clusterID++)
 7.    end if
 8. done; return labels
```

Procedure GPU-BFS (Node *v,* double *Epsilon,* double *minLines*, integer *clusterId*)
Performs a BFS on the graph *G* in order to discover clusters

```
 1. Initialize array F[1..graph.numNodes] with false values
 2. Initialize array V[1..graph.numNodes] with false values
 3. F[v] ← true // Put node v in the frontier
 4. while F has some node with a value of true do
 5.    GPU-BFS-Kernel(graph, Epsilon, minLines, F, V)
 6. done
 7. Bring the V array from the GPU to the host
 8. for each node n in the graph G do
 9.    if V[n] then
10.       label[n] ← clusterID; visited[n] ← true
11.    end if
12. done
```

Kernel GPU-BFS-Kernel (Graph *G,* double *epsilon,* double *minLines*, Array of boolean *frontier*, Array of boolean *visited*)
GPU Kernel that assists GPU-BFS in performing a BFS search

```
 1. if frontier[tID] then
 2.    frontier[tID] ← false; visited[tID] ← true
 3.    for each neighbor n of the node with identifier tID do
 4.       if not visited[n] then frontier[n] ← true end if
 5.    done
 6. end if
```

Algorithm 2. Pseudo-code of the G-TrajScan Algorithm

Porto has 1.7 million trajectories. Geolife occupies 700 MB and Porto occupies 1.8 GB. Each dataset is kept in the GPU's global memory.

B. *Hardware*

Our experiments were performed on a machine with Ubuntu 18.04 using CUDA 10. The hardware used was two Intel Xeon® 6136 processor and an NVIDIA Tesla V100 GPU with 16 GB.

C. *Parameters*

For DBSCAN, as the *minLines* parameter decreases and the *epsilon* parameter increases, the number of resulting clusters approaches one. We employ realistic measurements for both as default parameters for our tests. The parameters used in our experiments are presented Table I along with their default values indicated in bold.

TABLE I. DEFAULT PARAMETERS

Parameter Name	Values
Epsilon	10^{-4}, 10^{-3}, $\mathbf{10^{-2}}$, 0.1, 1, 10, 10^{2}
minLines	100, 200, 300, 400, 500, **600**
Num. of Trajectories	10^{1}, 10^{2}, 10^{3}, $\mathbf{10^{4}}$, 10^{5}
GPU Threads Per Block (GTraclus)	**512**
Num. of CPU Threads (MC-Traclus)	**24**

D. *Performance Measure*

In our experiments, the performance measure is the *total execution time* of the algorithm measured from the moment when it starts executing until the moment when the results are available in the host. Each query was run 5 times and we report the average of these 5 executions.

E. *Experimental Results*

In this section, we describe the impact of the parameters on the performance measures of the competing algorithms.

1) *Impact of the Number of Trajectories*

In this section, we describe the impact of the number of trajectories to cluster on the performance of each algorithm. Increasing the number of trajectories affected the processing time of both MC-Traclus and GTraclus, and while the MC-Traclus outperforms GTraclus for a small number of trajectories, GTraclus performs much better for large trajectory datasets.

Figure 1(a) shows the results of MC-Traclus and GTraclus executed on the Porto taxi dataset. In this figure we see that for fewer than 1,000 trajectories, the multicore CPU MC-Traclus performed faster than GTraclus, the latter performing 7.8X slower (5 ms versus 39 ms) for 100 trajectories. GTraclus however, started gaining advantage over MC-Traclus for datasets with over 1,000 trajectories, performing 5.5X faster for 10,000 trajectories and 22X faster for 100,000 trajectories. The reason for this is the GPU's parallel architecture is able to effectively support the independent distance calculations that G-DBSCAN utilizes. Every thread measures the parallel, perpendicular, and angular distances [8] independently and using hundreds of threads and overcomes the overheads produced by the memory transfers between main memory and device memory. In Figure 1(b), it is possible to see similar results for the Geolife dataset, where the performance improvement of GTraclus over MC-Traclus is even more pronounced than in the Porto dataset: GTraclus performs better starting from 100 trajectories instead of 1,000, and achieving up to 24X faster execution time than MC-Traclus for 100,000 trajectories. This is because each trajectory has fewer points compared to the Porto dataset, improving the execution time of the partitioning step on GPU, which is sensitive to trajectories with many points.

The partitioning stage on the GPU for 1,000 trajectories takes three times as much time as on MC-Traclus. At 10,000 and 100,000 trajectories it closed this gap and started to perform about as well as the original MC-Traclus partitioning implementation. This is because partitioning trajectories is a semi-sequential process. As mentioned earlier, the partitioning

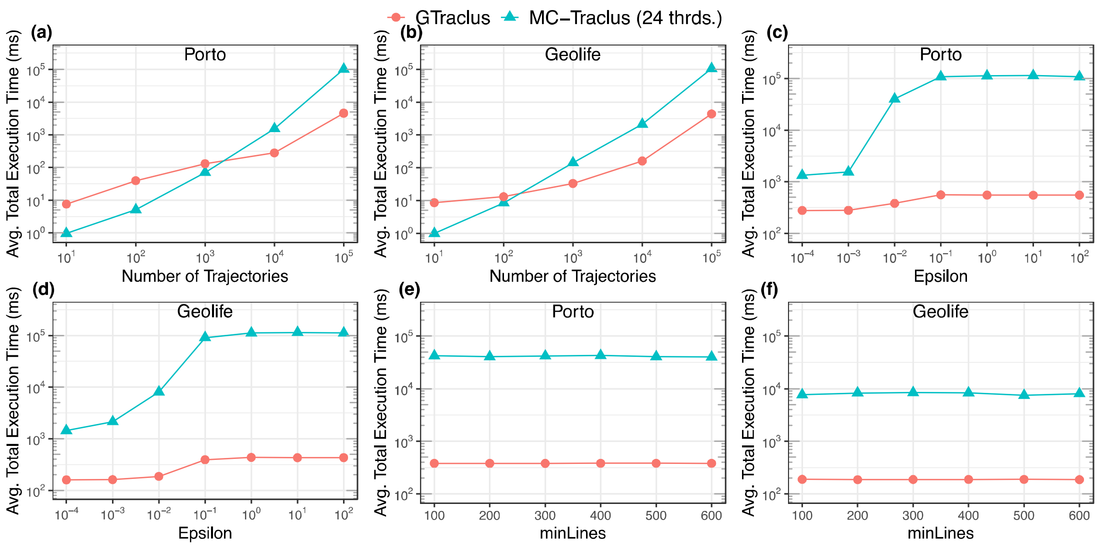

Figure 1. (a) Impact of the number of trajectories to cluster on the average total execution time in miliseconds in the Porto dataset. (b) Impact of the number of trajectories to cluster on the average total execution time in miliseconds in the Geolife dataset. (c) Impact of *Epsilon* on the average total execution time in miliseconds in the Porto dataset. (d) Impact of *Epsilon* on the average total execution time in miliseconds in the Geolife dataset. (e) Impact of *minLines* on the average total execution time in miliseconds in the Porto dataset. (f) Impact of *minLines* on the average total execution time in miliseconds in the Geolife dataset. In all experiments, 24 threads were used for MC-Traclus.

stage takes up less than 1% of the total time so a slowdown of three times as much, as was observed, is not as significant.

2) *Impacts of Epsilon and minLines*

In this experiment, we assess the impact of the *Epsilon* and *minLines* parameters on the performance of both algorithms. On one hand, we found that there was a significant impact of the *Epsilon* parameter on total execution time, as seen in Figure 1(c) and Figure 1(d). As *Epsilon* became larger, this led to the creation of a single large trajectory cluster and to an execution time increase of two orders of magnitude. This is due to the overhead in computing extremely large neighborhoods and performing very large graph searches for huge numbers of results. At the extreme parameter values tested, many segments that would be classified as noise are included in clusters, and many clusters which would be differentiated are included together. On the other hand, varying the *minLines* parameter within the range specified had little impact on the performance of GTraclus, as seen in Figure 1(e) and Figure 1(f). We conclude from these tests that GTraclus performance is resilient to extreme parameter values and clustering situations, as the change in performance was relatively even across these trials.

V. Conclusion and Future Work

In this paper, we presented a new GPU algorithm for local trajectory clustering based on the Minimum Description Length principle (MDL). Existing algorithms to solve the problem of local trajectory clustering based on this principle, like MC-Traclus, a multithreaded version of Traclus, do not scale. We addressed this gap by effectively parallelizing the distance computations between trajectories in the GPU. GTraclus provides the same clustering results as MC-Traclus but for large numbers of trajectories, starting around 100 to 1,000 trajectories, it results in up to 24X faster execution time than the comparable multi-core algorithm MC-Traclus. In the future, we would like to research the applications of GPU-based processing on fuzzy cluster membership of trajectories [18].

Acknowledgment

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

References

[1] Y. Zheng, "Location-Based Social networks: Users," in *Computing with Spatial Trajectories*, Z. Y and X. Zhou, Eds., Springer, 2011.

[2] Y. Zheng, X. Xie, W.-Y. Ma and others, "GeoLife: A collaborative social networking service among user, location and trajectory," *IEEE Data Eng. Bull.,* vol. 33, no. 2, 2010.

[3] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu and W.-Y. Ma, "Mining User Similarity Based on Location History," in *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2008.

[4] Y. Zheng, C. Licia, O. Wolfson and H. Yang, "Urban Computing: Concepts, Methodologies, and Applications," *ACM Transactions on Intelligent Systems and Technology (TIST),* vol. 5, no. 3, 2014.

[5] A. Ghose, Tap: Unlocking the Mobile Economy, MIT Press, 2018.

[6] M. D. Powell and S. D. Aberson, "Accuracy of United States Tropical Cyclone Landfall Forecasts in the Atlantic Basin (1976-2000)," *Bull. of the American Meteorological Society,* vol. 82, no. 12, 2001.

[7] M. J. Wisdon, N. J. Cimon, B. K. Johnson, E. O. Garton and J. W. Thomas, "Spatial Partitioning by Mule Deer and Elk in Relation to Traffic," in *Trans. of the North American Wildlife and Natural Resources Conf.*, 2004.

[8] J.-G. Lee, J. Han and K.-Y. Whang, "Trajectory Clustering: A Partition-and-Group Framework," in *ACM SIGMOD International Conference on Management of Data*, 2007.

[9] 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 *37th Annual International Symposium on Computer Architecture*, 2010.

[10] 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," *Journal of Parallel and Distributed Computing,* vol. 68, no. 10, 2008.

[11] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira and L. Damas, "Predicting Taxi-Passenger Demand Using Streaming Data," *IEEE Trans. on Intelligent Transportation Systems,* vol. 14, no. 3, 2013.

[12] "Programming Guide: Cuda Toolkit Documentation," NVIDIA Corporation, 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Accessed 11 October 2020].

[13] "CUDA C++ Best Practices Guide," NVIDIA Corporation, 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. [Accessed 11 October 2020].

[14] S. Gaffney and P. Smyth, "Trajectory Clustering with Mixtures of Regressions Models," in *ACM SIGKDD International Conference on Knowledge Discover and Data Mining*, 1999.

[15] Y. Zheng, "Trajectory Data Mining: An Overview," *ACM Transactions on Intelligent Systems and Technology,* 2015.

[16] S. Gaffney, A. Robertson, P. Smyth, S. Camargo and M. Ghil, "Probabilistic clustering of extratropical cyclones using regressions mixture models," *Climate Dynamics,* vol. 29, 2007.

[17] Z. Li, J.-G. Lee and J. Han, "Incremental Clustering of Trajectories," in *International Conference on Database Systems for Advanced Applications - Volume Part II.*, 2010.

[18] N. Pelekis, I. Kopanakis, E. E. Kotsifakos, E. Frentzos and Y. Theodoridis, "Clustering Uncertain Trajectories," *Knowledge Information Systems,* vol. 28, no. 1, 2011.

[19] G.-P. Roh and S.-w. Hwang, "NNCluster: An Efficient Clustering Algorithm for Road Network Trajectories," in *International Conference on Database Systems for Advanced Applications*, 2010.

[20] Z. Deng, Y. Hu, M. Zhu, X. Huang and B. Du, "A scalable and fast OPTICS for clustering trajectory big data," *Cluster Computing,* vol. 18, no. 2, 2015.

[21] J. Gudmundsson and N. Valladares, "A GPU approach to subtrajectory clustering using the Fréchet distance," in *ACM SIGSPATIAL*, 2012.

[22] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira and L. Rocah, "G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering," *Procedia Computer Science,* vol. 18, 2013.

[23] M. Ester, H.-P. Kriegel, J. Sander and X. Xu, "A Density-based Algorithm for Discovering Clustering in Large Spatial Databases with Noise," in *International Conference on Knowledge Discover and Data Mining*, 1996.

[24] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU using CUDA," *Proceedings of the International Conference on High Performance Computing,* 2007.

[25] E. Leal and L. Gruenwald, "DynMDL: A Parallel Trajectory Segmentation Algorithm," *IEEE Int'l. Congress on Big Data,* 2018.