Distributed and Parallel Databases (2023) 41:467–488
https://doi.org/10.1007/s10619-023-07429-x

GTraclus: a novel algorithm for local trajectory clustering on GPUs

Hamza Mustafa[1] · Clark Barrus[2] · Eleazar Leal[1] · Le Gruenwald[2]

Accepted: 20 April 2023 / Published online: 13 May 2023

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 24× faster execution time when compared against MC-Traclus.

Keywords Trajectory · Clustering · GPU · Spatial data · Parallel computing

✉ Eleazar Leal
eleal@d.umn.edu

Hamza Mustafa
musta067@d.umn.edu

Clark Barrus
clark.barrus@ou.edu

Le Gruenwald
ggruenwald@ou.edu

[1] Department of Computer Science, University of Minnesota Duluth, Duluth, MN, USA

[2] School of Computer Science, University of Oklahoma, Norman, OK, USA

1 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 analyzing 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 its wireless network, can send personalized advertising information to customers based on their paths inside the mall [5].

An important consideration for trajectory clustering is whether the elements to be clustered are entire trajectories, in which case we say that we perform global trajectory clustering, or whether they are sub-trajectories, which gives rise to the problem of local trajectory clustering. In many applications, clustering entire trajectories may not provide significant 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 using trajectories for predicting a hurricane's landfall, meteorologists are more interested in clustering hurricane behaviors near the coastline or at 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, even when compared 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. The MDL principle allows compressing trajectories by minimizing the approximation error and the total number of points resulting from the compression. 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.

A preliminary version of this paper was presented in 2021 at the Joint International Workshop on Big Data Management on Emerging Hardware and Data Management on Virtualized Active Systems (HardBD & Active), which is a part of IEEE International Conference on Data Engineering Workshops (ICDEW) [12]. This paper extends the workshop version by providing the following additional details and experiments: (1) we perform a study of the impact of the different stages of GTraclus on its overall execution time performance; (2) we extend our proposed GTraclus algorithm with a linear work complexity GPU algorithm for trajectory cluster expansion; (3) we evaluate the impact of this algorithm on the overall execution time performance; and (4) we revise the presentation, adding more related work.

The remainder of this paper is organized as follows: Sect. 2 presents background concepts and related work; Sect. 3 contains the description of the proposed GTraclus algorithm; Sect. 4 contains a thorough performance analysis; and finally, Sect. 5 presents conclusions and future work.

2 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.

2.1 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 [13], 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 separated from the host's, which makes it necessary to send all input data through the PCIe bus to the GPU before any processing can take place in the latter, 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) low global memory bandwidth relative to the number of threads. Because of the large number of threads contending for global memory access within a GPU, it is desirable to group those memory requests into as few memory transactions as possible. This can be done by the memory coalescing unit if those accesses respect memory coalescing, i.e., if consecutive threads access adjacent memory locations [14]; 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.

The problem of trajectory clustering on GPUs is related to that of arbitrary data clustering on the same architecture [15, 16]. In particular, our proposed algorithm, GTraclus, is similar to general-purpose density-based clustering algorithms. Nonetheless, existing work on GPU algorithms for the latter does not consider the spatio-temporal correlation of trajectory data. This correlation impacts how this data type is arranged in the GPU's global memory and the calculation of the similarity measure. There are multiple works devoted to studying algorithms for density-based clustering on GPUs: Thapa et al.'s algorithm [17], CUDA-DClust [18], CUDA-DClust+ [19], CudaSCAN [20], Prokopenko's algorithm [21], and GPU-INSCY [15, 16]. Our previous work [22] provides an experimental comparison among several of these GPU algorithms for general data clustering.

2.2 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 [23–25].

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 it 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 [26], CenTra-I-FCM [27], NNCluster [28], and TSST-HDBC [29], none 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 [30], a density-based point clustering algorithm for global trajectory clustering, not for local trajectory clustering like ours, and the work by Gudmundsson and Valladares [31], which, unlike GTraclus, finds clusters of similar sub-trajectories within a single trajectory and makes use of the Fréchet distance and not MDL.

3 Proposed algorithm

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

3.1 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. Its pseudo-code is presented in Algorithm 1. 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. This stage is called in Line 1 of Algorithm 1.

The *grouping stage* 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. This stage is called in Line 2 of Algorithm 1.

We store this reachability graph in Compressed Sparse Row (CSR) format [32], which has a low memory overhead that makes it very popular [33]. This format stores a graph with two arrays: *vertex* and *edge*. The *vertex* array satisfies the following property: for every integer i, an id of a vertex, $vertex[i]$ contains the offset in *edge* such that all elements in $edge[vertex[i]]$ up to and including $edge[vertex[i+1]-1]$ are the ids of the vertices adjacent to i.

Algorithm 1 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: $segments \leftarrow GPU\text{-}Partition(D)$ ▷ Partitioning Stage
2: $labels \leftarrow G\text{-}TrajScan(segments, Epsilon, minLines)$ ▷ Grouping Stage
3: **return** $(segments, labels)$

3.2 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 $T = T_1, T_2, \ldots, T_{size(T)}$, 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 $L(H) + L(D \mid H)$ is minimized, where $L(H)$ and $L(D \mid H)$ are defined as:

$$L(H) = \sum_{j=0}^{l-2} \log_2 \left(Length(T_{c_j} T_{c_{j+1}}) \right)$$

$$L(D \mid H) = \sum_{j=0}^{l-2} \sum_{k=c_j}^{c_{j+1}-1} \left[\log_2 \left(d_\perp (T_{c_j} T_{c_{j+1}}, T_k T_{k+1}) \right) + \log_2 \left(d_\theta (T_{c_j} T_{c_{j+1}}, T_k T_{k+1}) \right) \right],$$

where $T_{c_j} T_{c_{j+1}}$ denotes the segment from T_{c_j} to $T_{c_{j+1}}$ and $Length(T_{c_j} T_{c_{j+1}})$ is the Euclidean distance between its endpoints, $d_\perp$ is the orthogonal distance [26] 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, and d_θ is the *angular distance* between those two segments and is defined as:

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

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 2, returns the array *dSegs* containing the number of segments for each trajectory.

Algorithm 2 GPU-Partition (Trajectory Set D)
Each trajectory in D is partitioned in parallel according to the MDL principle

```
1: for trajectory t ∈ D do in parallel
2:     dSegs[t] ← CountPartitions(t)
3: end for
4: dSegs ← InclusivePrefixSum(dSegs)
5: for trajectory t ∈ D do in parallel
6:     segments ← FillPartitions(t, dSegs)
7: end for
8: return segments
```

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 2. 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.

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.

As mentioned, the GPU-Partition algorithm (Algorithm 2) is parallelized by having GPU threads partition different trajectories. The disadvantage of this approach is that warps load points located in non-adjacent memory locations; this decision stems from the fact that trajectories are not guaranteed to have the same number of points. However, as we will see in Sect. 4.5.3, this potential lack of global memory coalescing does not have a significant impact on the overall execution time of GTraclus for a large number of trajectories because the partitioning stage does not necessarily dominate the execution time of our algorithm.

3.3 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 [34], 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.

The grouping stage has two sub-phases: making of the density connected graph of segments and the identification of segment clusters. We know explain each of these.

Algorithm 3 GPU-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: $graph \leftarrow Make\text{-}Graph(D, Epsilon, minLines)$
2: $labels \leftarrow Identify\text{-}Clusters(graph, Epsilon, minLines)$
3: **return** $labels$

3.3.1 Making the density connected graph

In Line 1 of the procedure G-TrajScan in Algorithm 3, 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 1–8 of the Make-Graph function in Algorithm 4 count the number of segments within an *Epsilon* distance of each segment, then Line 9 computes a prefix sum over the number of neighbors of each segment with the purpose of allocating space for the CSR representation of the graph. Lines 10–15 then compute the *edge_array* by checking in parallel which segments are within an *Epsilon* distance for each node.

Algorithm 4 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**
2: $numNeighbors[p] \leftarrow 0$
3: **end for**
4: **for** each pair (p, q) of segments in D **do in parallel**
5: **if** $dist(p, q) < Epsilon$ **then**
6: $numNeighbors[p]$++
7: **end if**
8: **end for**
9: $edge\_array \leftarrow ExclusivePrefixSum(numNeighbors)$
10: **for** each pair (p, q) of segments in D **do in parallel**
11: **if** $dist(p, q) < Epsilon$ **then**
12: Add q to the list $vertex\_array$ of neighbors of p and vice versa
13: **end if**
14: **end for**
15: **return** new Graph($vertex\_array$, $edge\_array$)

The Make-Graph Algorithm (Algorithm 4) is designed to exploit global memory coalescing because all segments of all trajectories are represented as four arrays: $start_x, start_y, end_x, end_y$, allowing the threads in a warp to access their own points in a coalesced manner in Lines 4–8 and 10–14; the fact that segments are represented using those four arrays is not purposely shown in the pseudo-code of Algorithm 4 so that the presentation is cleaner. Using the array *numNeighbors* to store the number of neighbors of each segment allows coalesced global memory accesses on two different occasions: first, after the threads have computed these values on their own and write to the *numNeighbors* array (after Line 8), and second, during the exclusive prefix sum computation (Line 9). The Make-Graph is load balanced in Lines 1–9, except during the writing of the neighbors (Lines 10–14) because different segments will have different numbers of neighbors.

3.3.2 Identifying the segment clusters

Once the reachability graph is created, a GPU-based BFS is initiated in Line 2 of the GPU-TrajScan procedure in Algorithm 3 by calling the Identify-Clusters function in Algorithm 5. This algorithm begins with marking all nodes as not visited (Lines 2–4 of Algorithm 5). Then, in Lines 5–10, it starts a BFS search from each unvisited core segment, where a core segment is one with at least *minLines* many segments within a segment distance of *Epsilon*.

Algorithm 5 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 all node v do
3:      visited[v] ← false
4:  end for
5:  for each node v ∈ G do
6:      if visited[v] = false and v is a core segment then
7:          visited[v] ← true
8:          labels[v] ← clusterID
9:          GPU-BFS(v, G, Epsilon, minLines, clusterID); clusterID++
10:     end if
11: end for
12: return labels
```

During a BFS search from a core segment, shown in Algorithm 6, each thread goes into the adjacency lists of density-connected line segments and marks them as visited (Lines 5–11 in the Identify-Clusters procedure of Algorithm 5). All density-connected line segments [35] 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 Algorithm 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.

In Lines 1–3 of Algorithm 6, we initialize two arrays, *frontier* and *V* of length *G.numNodes* with *false* values, indicating that no vertex is in the frontier and no vertex has been visited, and then place the vertex *v* in the *frontier* before calling, in Lines 4–6, the GPU-BFS-Kernel of Algorithm 7.

This kernel has different threads check in parallel if *each* node in the graph belongs to the *frontier*, and if it does, it removes the node from the *frontier*, marks it as *visited* (Lines 1–3 of Algorithm 7), and adds its unvisited neighbor nodes to the *frontier* (Lines 4–9). In Algorithm 6, 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 the global memory.

Algorithm 6 GPU-BFS (Graph G, Node v, double $Epsilon$, double $minLines$, integer $clusterID$)
Performs a BFS on the graph G in order to discover clusters

```
1: Initialize local array frontier[1...G.numNodes] with false values
2: Initialize local array V[1...G.numNodes] with false values
3: frontier[v] ← true // Put node v in the frontier
4: while frontier has some node with a value of true do
5:     GPU-BFS-Kernel(G, Epsilon, minLines, frontier, V)
6: end while
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
11:        visited[n] ← true
12:    end if
13: end for
```

Algorithm 7 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[threadID] then
2:     frontier[threadID] ← false
3:     visited[threadID] ← true
4:     for each neighbor n of the node with identifier threadID do
5:         if visited[n] = false and n is a Core segment then
6:             frontier[n] ← true
7:         end if
8:     end for
9: end if
```

3.4 Further enhancements to GTraclus

As can be seen from our description in Sect. 3.3, Algorithm 6, originally presented in our paper [12], has quadratic work complexity because *every GPU thread* checks if its assigned elements belong to the frontier of the graph in Line 1 of Algorithm 7, regardless of the size of the frontier. In this section, we present an improved, linear complexity version of this algorithm: GPU-BFS-Linear (see Algorithm 8) and its associated kernel, GPU-BFS-Linear-Kernel (see Algorithm 9), adapted from [36, 37]. These two algorithms are drop-in replacements for Algorithms 6 and 7. Algorithm 8 initializes two queues, $queue_1$ and $queue_2$, and a V array, all with a capacity equal to the number of nodes in the segment graph (Lines 1–2 of Algorithm 8). Then, it places the starting node v in $queue_1$ before calling the linear algorithm GPU-BFS-Linear-Kernel (Line 11). This kernel, unlike the quadratic one, does not check each node for containment in the frontier; instead, each GPU thread takes an element v from the queue (Line 2 of Algorithm 9) and adds to the *next_queue* all the unvisited core neighbors of v (Lines 4–9 of Algorithm 9). This last operation is done with an atomic add to make sure that the writes to the globally shared *next_queue* are coordinated. The *next_queue* will be the *queue* used the next time that the GPU-BFS-Linear-Kernel is called. This alternation between *queue* and *next_queue* is accomplished by keeping track of the parity of the number of calls to the GPU-BFS-Linear-Kernel.

This kernel addresses the issue of low global memory bandwidth relative to the number of threads by having threads by consecutive thread ids access adjacent elements of the *queue* array (Line 2 in Algorithm 9); nonetheless, accessing the neighbors of those nodes might not guarantee global memory coalescing.

Algorithm 8 GPU-BFS-Linear (Graph G, Node v, double $Epsilon$, double $minLines$, integer $clusterId$)
Performs a BFS on the graph G in order to discover clusters

```
1: Create two empty queues queue_1, queue_2 with capacity G.numNodes in the GPU
2: Initialize local array V[1 ... G.numNodes] with false values
3: level ← 0
4: queue_1.insert(v) // Put node v in queue_1
5: repeat
6:     if level mod 2 = 0 then
7:         queue, next_queue ← queue_1, queue_2
8:     else
9:         queue, next_queue ← queue_2, queue_1
10:    end if
11:    GPU-BFS-Linear-Kernel(G, Epsilon, minLines, V, queue, next_queue)
12:    level ← level + 1
13: until next_queue = ∅
14: for each node n in the graph G do
15:    if V[n] then
16:        label[n] ← clusterID; visited[n] ← true
17:    end if
18: end for
```

Algorithm 9 GPU-BFS-Linear-Kernel (Graph G, double $epsilon$, double $minLines$, Array of boolean $visited$, Queue of Node $queue$, Queue of Node $next\_queue$)
GPU Kernel that assists GPU-BFS in performing a BFS search

```
1: if threadID < queue.size then
2:     v ← queue[threadID]
3:     visited[v] ← true
4:     for each neighbor n of the node v do
5:         if visited[n] = false and n is a Core segment then
6:             position ← atomicAdd(next_queue.size, 1)
7:             next_queue[position] ← n
8:         end if
9:     end for
10: end if
```

4 Performance analysis

In this section, we describe the datasets, the 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.

4.1 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 in China, while the Porto trajectories log taxis as they move through the city of Porto in Portugal. Both datasets contain a considerable amount of data; Geolife has 17,621 trajectories, which were broken down into over 100,000 trajectories after stay-point detection [24], and Porto has 1.7 million trajectories. Geolife occupies 700 MB and Porto 1.8 GB. Each dataset is kept in the GPU's global memory.

We have chosen these two datasets because they are well-known and have been used in recent years in research studying the scalability of clustering algorithms on Apache Spark [38] and GPUs [19]. This is because these are real-life trajectory datasets with significant noise and skew that test the limits of indexing and clustering algorithms.

4.2 Hardware

Our experiments were performed on a machine with Ubuntu 18.04 using CUDA 10 and compiled with gcc v11.3 with `-O3` optimizations. The hardware used was two Intel Xeon®6136 processors, an Nvidia A4500 GPU with 20 GB, and an Nvidia Tesla V100 GPU with 16 GB.

4.3 Parameters

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 1 along with their default values indicated in bold.

4.4 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

Table 1 Experimental 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	**24**

The default parameter values are bolded

the results are available in the host. Each query was run five times, and we report the average of these five executions. In these experiments, the standard deviations of the execution times are not shown, because they are very small and not noticeable in the figures.

4.5 Experimental results

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

4.5.1 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 affects the processing time of both MC-Traclus and GTraclus, and while MC-Traclus outperforms GTraclus for a small number of trajectories, GTraclus performs much better for large trajectory datasets, regardless of whether GTraclus runs on the Nvidia A4500 (A4500) or the Nvidia V100 (V100).

Figure 1a shows the results of MC-Traclus and GTraclus executed on the Geolife taxi dataset. In this figure, it is possible to see that the performance improvement of GTraclus (both V100 and A4500) over MC-Traclus is more pronounced than for the Porto dataset: GTraclus (V100) performs better starting from 100 trajectories instead of 1000, achieving up to 186× faster execution time than MC-Traclus for 100,000 trajectories. This is because the trajectories in the Geolife dataset have fewer points than in the Porto dataset, improving the execution time of the partitioning step on GPU, which is sensitive to trajectories with many points. A similar behavior is seen for GTraclus (A4500) except that it performs better than MC-Traclus starting from around 700 trajectories, achieving up to 28× faster execution time than MC-Traclus for 100,000 trajectories. The gap between the execution times of GTraclus (V100) and GTraclus (A4500) with 100,000 trajectories is around 6.7×.

Figure 1b shows the results of MC-Traclus and GTraclus executed on the Porto taxi dataset. The results are similar to those seen with the Geolife dataset. In this figure, we see that for fewer than 1000 trajectories, the multicore CPU MC-Traclus performed faster than GTraclus (V100), the latter performing 7.8× slower (5 ms versus 39 ms) for 100 trajectories. However, GTraclus (V100) started having an advantage

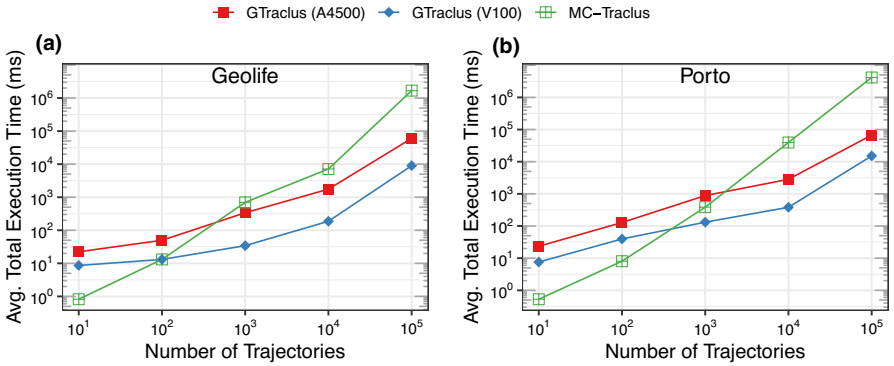

Fig. 1 **a** Impact of the number of trajectories to cluster on the average total execution time in milliseconds for the Geolife dataset. **b** Impact of the number of trajectories to cluster on the average total execution time in milliseconds for the Porto dataset. In all experiments, 24 threads were used for MC-Traclus

over MC-Traclus for datasets with over 1000 trajectories, performing 5.5× faster for 10,000 trajectories and 22× faster for 100,000 trajectories. A similar behavior is observed between GTraclus (A4500) and MC-Traclus, except that it takes slightly over 1000 trajectories for GTraclus (A4500)'s execution time to improve upon MC-Traclus's. The reason for this is that the GPU's parallel architecture can effectively support the independent distance calculations of GTraclus. 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.

The partitioning stage on the GPU for 1000 trajectories takes three times longer than with MC-Traclus. At 10,000 and 100,000 trajectories, it closes this gap and starts to perform about as well as the original MC-Traclus partitioning implementation. This is because partitioning trajectories is semi-sequential.

In our experiments, we verified that one of the main challenges for scalable trajectory clustering on GPUs is fitting the data structure used by the algorithm during the Make-Graph stage, which requires more memory (around 12 GB for Porto) than the data itself. GTraclus addresses this issue in part thanks to its CSR representation (See Sect. 3.3), which avoids using explicit pointers, keeping a compact data structure. However, to scale several orders of magnitude beyond the number of trajectories in these datasets, future research needs to be done to eliminate the assumption that the data structures must fit entirely in global memory.

4.5.2 Impacts of *Epsilon* and *minLines*

In this experiment, we assess the impact of the *Epsilon* and *minLines* parameters on the performance of the competing algorithms. On one hand, we find that there is a significant impact of the *Epsilon* parameter on the total execution time, as seen in Fig. 2a and b. As *Epsilon* becomes larger, this leads to the creation of a

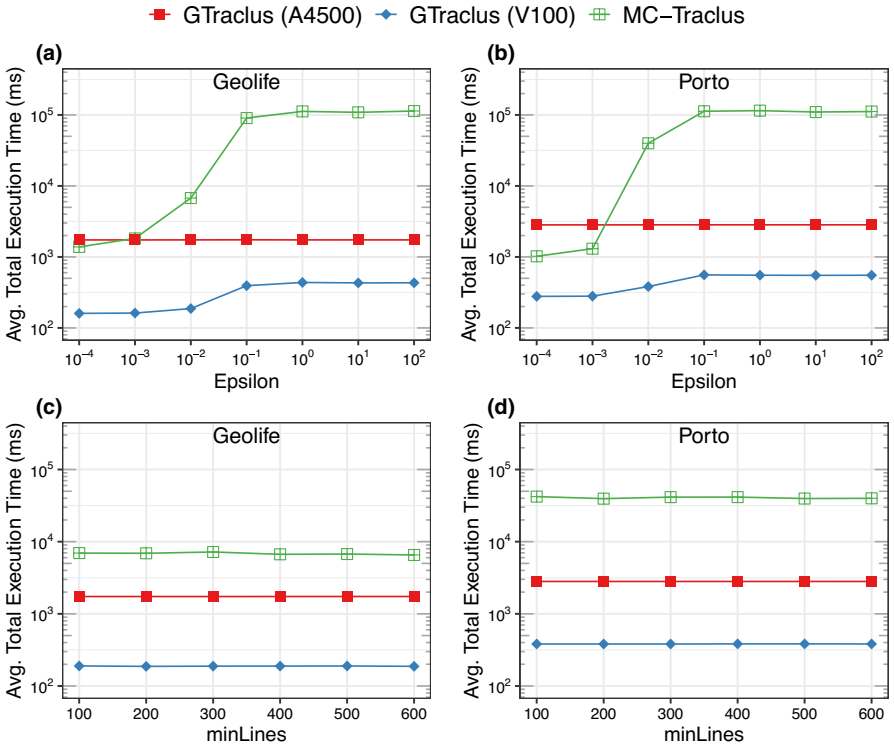

Fig. 2 **a** Impact of *Epsilon* on the average total execution time in milliseconds for the Geolife dataset. **b** Impact of *Epsilon* on the average total execution time in milliseconds for the Porto dataset. **c** Impact of *minLines* on the average total execution time in milliseconds for the Geolife dataset. **d** Impact of *minLines* on the average total execution time in milliseconds for the Porto dataset. In all experiments, 24 threads were used for MC-Traclus

single large trajectory cluster and to an execution time increase of up to 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 merged. In Fig. 2a and b we also see that for low values of *Epsilon*, the performance of GTraclus (A4500) becomes competitive with that of MC-Traclus; this is expected because the performance of the A4500 is around 6× slower than the V100, corresponding to shifting upwards the execution time curve of the V100; the net impact of low *Epsilon* values is that it increases the number of trajectories required for the A4500's execution time to improve upon MC-Traclus's. On the other hand, varying the *minLines* parameter within the range specified has little impact on the performance of GTraclus, as seen in Fig. 2c and d. We conclude from these tests that GTraclus' performance is resilient to extreme parameter values and clustering situations, as the change in performance is relatively even across these trials.

4.5.3 Stage time distribution

In this experiment, we measure the rate between the execution time of the individual stages of GTraclus to the overall execution time of this algorithm when run under the default experimental parameters, shown in Table 1. We have used the execution times obtained with the V100 for this section.

The results of our experiments are presented in Table 2. There we see that for the Geolife dataset, the smallest of our test sets, roughly 10% of the total execution time comes from the trajectory partitioning stage, while the construction of the graph (Make-Graph, presented in Algorithm 4) and its BFS traversal (Identify-Clusters, presented in Algorithm 5) each contributes around 45% of the total execution time. In the Porto dataset, partitioning takes around 46% of the total execution time, while constructing and traversing the graph each takes around 27% of the execution time. The discrepancy between the partitioning percentages across these datasets stems from the fact that the Porto dataset gives rise to many more trajectory segments (3.87×) than Geolife; therefore, GTraclus partitioning does more work for Porto than for the other dataset. This is because these datasets have different spatio-temporal distributions, with the Porto dataset containing trajectories that are harder to "compress", according to MDL, than those of Geolife. These results indicate that for datasets with hard-to-compress trajectories, partitioning dominates the total execution time, while for other datasets, the clustering algorithms, Make-Graph and Identify-Clusters, dominate the execution time.

Figure 3a and b present the impact of the number of trajectories on the execution times of the stages of GTraclus using the Geolife and the Porto datasets, respectively. The grouping time corresponds to the sum of the execution time of the Make-Graph and Identify-Clusters sub-stages. In both datasets, we observe a similar behavior: the contribution of the grouping time to the total execution time is negligible when trajectories are few, but as the number of trajectories increases (beyond 1000 and 10,000 in Geolife and Porto, respectively), the grouping time dominates the total execution time.

4.5.4 Impact of the GPU BFS traversal algorithm

In this experiment, we investigate the impact of the GPU BFS traversal algorithm on the execution time performance of GTraclus (V100). We compare two BFS traversal algorithms: the quadratic and the linear. The first, Algorithm 6, was the one used in the workshop version of this paper [12]; the second, Algorithm 8, is the one described in Sect. 3.4. As can be seen from their pseudo-codes, one of the key

Table 2 Stage time distribution

GTraclus stage	Geolife (%)	Porto (%)
Trajectory partitioning	9.72	45.86
Make-graph	45.04	26.92
Identify-clusters	45.26	27.20

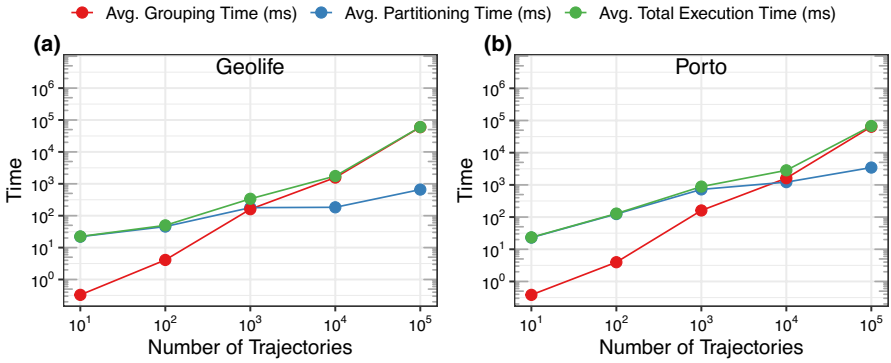

Fig. 3 **a** Impact of the number of trajectories on the execution time in milliseconds of each GTraclus stage for the Geolife dataset. Grouping Time is the sum of the time of Make-Graph and Identify-Clusters. **b** Impact of the number of trajectories on the average execution time in milliseconds of each GTraclus stage for the Porto dataset

differences between them stems from the fact that the linear version uses a queue implemented as an array to contain all the vertices that are in the frontier of the segment graph, while the quadratic version does not do this and instead uses a boolean array of length $|V|$, called *frontier*, where V is the set of all vertices, indicating whether each vertex has been visited or not; every GPU thread is assigned a different vertex in V to test if it is in the frontier or not. In the linear algorithm, every GPU thread is assigned a different vertex in *frontier*, instead of a different vertex of V.

From the experiments we performed in our datasets, the performance improvement of the linear BFS algorithm over the quadratic one is negligible because the sizes of the datasets are not large enough for the difference to show. This can be seen in Fig. 4, which shows the difference between the average total execution times when using the quadratic and the linear BFS traversal as a function of the number of trajectories. Even when the total execution time is highest for both datasets (i.e., when using 100,000 trajectories), the difference between the execution times represents only a small fraction of the total execution time of GTraclus: around 100 ms when the total execution time is in the order of 10,000 ms. However, it is evident that the linear BFS algorithm should be the preferred choice, especially for dealing with even larger datasets than the ones in this study, since the effort required to implement it is minimal when compared to that of the quadratic BFS algorithm.

5 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,

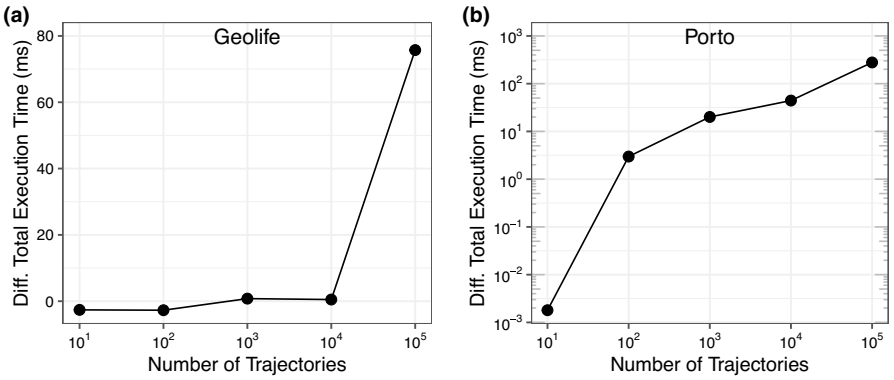

Fig. 4 **a** Impact of the number of trajectories on the difference between the average total execution times in milliseconds of GTraclus when using the quadratic versus the linear BFS traversal with the Geolife dataset. **b** Impact of the number of trajectories on the difference between the average total execution times in milliseconds of GTraclus when using the quadratic versus the linear BFS traversal with the Porto dataset

starting around 100 to 1000 trajectories, it results in up to 24× faster execution time than MC-Traclus. Our experiments suggest that for datasets with hard-to-compress trajectories, the trajectory partitioning stage dominates the total execution time, while for other datasets, the cluster discovery algorithms dominate the execution time. Our experiments also show that the GPU BFS traversal algorithm used for discovering clusters does not have a noticeable impact on the execution time performance of our proposed algorithm. GTraclus assumes that the data fits in the global memory of the GPU. It also assumes that the neighbor graph, which in our experiments occupied more space than the data, also fits along in global memory. A potential avenue for future research is studying new algorithms that do not make these two assumptions. Another possible future work is to research the applications of GPU-based processing on fuzzy cluster membership of trajectories [27].

Acknowledgements This work is supported in part by the National Science Foundation under Grant Nos. 1302439 and 1302423.

Author contributions All authors contributed equally to the manuscript.

Declarations

Competing interests The authors declare no competing interests.

References

1. Zheng, Y.: Location-based social networks: users. In: Zheng, Y., Zhou, X. (eds.) Computing with Spatial Trajectories (2011). https://doi.org/10.1007/978-1-4614-1629-6_8
2. Zheng, Y., Xie, X., Ma, W.: Geolife: a collaborative social networking service among user, location and trajectory. IEEE Data Eng. Bull. **33** (2010)

3. Li, Q., Zheng, Y., Xie, X., Chen, Y., Liu, W., Ma, W.-Y.: Mining user similarity based on location history. In: Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. GIS '08. Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1463434.1463477
4. Zheng, Y., Capra, L., Wolfson, O., Yang, H.: Urban computing: Concepts, methodologies, and applications. ACM Trans. Intell. Syst. Technol. **5** (2014). https://doi.org/10.1145/2629592
5. Ghose, A.: Tap: Unlocking the Mobile Economy (2017)
6. Powell, M.D., Aberson, S.D.: Accuracy of United States tropical cyclone landfall forecasts in the Atlantic basin (1976-2000). Bull. Am. Meteorol. Soc. **82** (2001). https://doi.org/10.1175/1520-0477(2001)082<2749:AOUSTC>2.3.CO;2
7. Wisdom, M.J., Cimon, N.J., Johnson, B.K., Garton, E.O., Thomas, J.W.: Spatial partitioning by mule deer and elk in relation to traffic (2004)
8. Lee, J.-G., Han, J., Whang, K.-Y.: Trajectory clustering: a partition-and-group framework. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. SIGMOD '07, pp. 593–604. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1247480.1247546
9. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. (2010). https://doi.org/10.1145/1815961.1816021
10. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. **68** (2008). https://doi.org/10.1016/j.jpdc.2008.05.014
11. Moreira-Matias, L., Gama, J., Ferreira, M., Mendes-Moreira, J., Damas, L.: Predicting taxi-passenger demand using streaming data. IEEE Trans. Intell. Transp. Syst. **14** (2013). https://doi.org/10.1109/TITS.2013.2262376
12. Mustafa, H., Barrus, C., Leal, E., Gruenwald, L.: Gtraclus: A local trajectory clustering algorithm for GPUS. In: 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW), pp. 30–35 (2021). https://doi.org/10.1109/ICDEW53142.2021.00013
13. Nvidia: Cuda C++ Programming Guide Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed 11 Oct 2020
14. Nvidia: Cuda C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Accessed 11 Oct 2020
15. Jørgensen, J.R., Scheel, K., Assent, I., Pathak, A.R., Elster, A.C.: GPU-FAST-PROCLUS: a fast GPU-parallelized approach to projected clustering. In: EDBT, pp. 2–196 (2022). https://doi.org/10.48786/edbt.2022.09
16. Jørgensen, J.R., Scheel, K., Assent, I.: GPU-INSCY: A GPU-parallel algorithm and tree structure for efficient density-based subspace clustering. In: EDBT, pp. 25–36 (2021). https://doi.org/10.5441/002/edbt.2021.04
17. Thapa, R.J., Trefftz, C., Wolffe, G.: Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases. In: 2010 IEEE International Conference on Electro/Information Technology, pp. 1–5 (2010). https://doi.org/10.1109/EIT.2010.5612134
18. Böhm, C., Noll, R., Plant, C., Wackersreuther, B.: Density-based clustering using graphics processors. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, pp. 661–670 (2009). https://doi.org/10.1145/1645953.1646038
19. Poudel, M., Gowanlock, M.: CUDA-DClust+: Revisiting early GPU-accelerated DBSCAN clustering designs. In: 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), pp. 354–363 (2021). https://doi.org/10.1109/HiPC53243.2021.00049
20. Loh, W.-K., Yu, H.: Fast density-based clustering through dataset partition using graphics processing units. Inf. Sci. **308**, 94–112 (2015). https://doi.org/10.1016/j.ins.2014.10.023
21. Prokopenko, A., Lebrun-Grandié, D., Arndt, D.: Fast tree-based algorithms for DBSCAN on GPUS. CoRR arXiv:2103.05162 (2021)
22. Mustafa, H., Leal, E., Gruenwald, L.: An experimental comparison of GPU techniques for DBSCAN clustering. In: 2019 IEEE International Conference on Big Data (Big Data), pp. 3701–3710 (2019). https://doi.org/10.1109/BigData47090.2019.9006169
23. Gaffney, S., Smyth, P.: Trajectory clustering with mixtures of regression models. In: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

KDD '99, pp. 63–72. Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/312129.312198
24. Zheng, Y.: Trajectory data mining: an overview. ACM Trans. Intell. Syst. Technol. **6**(3) (2015). https://doi.org/10.1145/2743025
25. Gaffney, S.J., Robertson, A.W., Smyth, P., Camargo, S.J., Ghil, M.: Probabilistic clustering of extratropical cyclones using regression mixture models. Clim. Dyn. **29** (2007). https://doi.org/10.1007/s00382-007-0235-z
26. Li, Z., Lee, J.G., Li, X., Han, J.: Incremental Clustering for Trajectories, vol. 5982 LNCS (2010). https://doi.org/10.1007/978-3-642-12098-5_3
27. Pelekis, N., Kopanakis, I., Kotsifakos, E.E., Frentzos, E., Theodoridis, Y.: Clustering uncertain trajectories. Knowl. Inf. Syst. **28** (2011). https://doi.org/10.1007/s10115-010-0316-x
28. Roh, G.-P., Hwang, S.-W.: Nncluster: An efficient clustering algorithm for road network trajectories. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) Database Systems for Advanced Applications, pp. 47–61. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-12098-5_4
29. Zhang, X., Niu, X., Fournier-Viger, P., Wang, B.: Two-stage traffic clustering based on HNSW. In: Advances and Trends in Artificial Intelligence. Theory and Practices in Artificial Intelligence: 35th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2022, Kitakyushu, Japan, July 19–22, 2022, Proceedings, pp. 609–620. Springer, Berlin (2022). https://doi.org/10.1007/978-3-031-08530-7_51
30. Deng, Z., Hu, Y., Zhu, M., Huang, X., Du, B.: A scalable and fast optics for clustering trajectory big data. Cluster Comput. **18** (2015). https://doi.org/10.1007/s10586-014-0413-9
31. Gudmundsson, J., Valladares, N.: A GPU approach to subtrajectory clustering using the fréchet distance. IEEE Trans. Parallel Distrib. Syst. **26** (2015). https://doi.org/10.1109/TPDS.2014.2317713
32. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the gpu using cuda. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) High Performance Computing—HiPC 2007, pp. 197–208. Springer, Berlin (2007). https://doi.org/10.1007/978-3-540-77220-0_21
33. Min, S.W., Mailthody, V.S., Qureshi, Z., Xiong, J., Ebrahimi, E., Hwu, W.: Emogi: Efficient memory-access for out-of-memory graph-traversal in GPUS. Proc. VLDB Endow. **14**(2), 114–127 (2020). https://doi.org/10.14778/3425879.3425883
34. Andrade, G., Ramos, G., Madeira, D., Sachetto, R., Ferreira, R., Rocha, L.: G-dbscan: A GPU accelerated algorithm for density-based clustering. Procedia Comput. Sci. **18**, 369–378 (2013). https://doi.org/10.1016/j.procs.2013.05.200. 2013 International Conference on Computational Science
35. Ester, M., Kriegel, H.-P., Sander, J., Xu, X.: 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, pp. 226–231. AAAI Press, Portland, Oregon (1996). https://doi.org/10.5555/3001460.3001507
36. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. SIGPLAN Not. **47**(8), 117–128 (2012). https://doi.org/10.1145/2370036.2145832
37. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '12, pp. 117–128. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2145816.2145832
38. Song, H., Lee, J.-G.: RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1173–1187 (2018). https://doi.org/10.1145/3183713.3196887