Distance Threshold Similarity Searches on Spatiotemporal Trajectories using GPGPU

Michael Gowanlock
Dept. of Information and Computer Sciences and
NASA Astrobiology Institute
University of Hawai'i, Honolulu, HI, U.S.A.
Email: gowanloc@hawaii.edu

Henri Casanova
Dept. of Information and Computer Sciences
University of Hawai'i, Honolulu, HI, U.S.A.
Email: henric@hawaii.edu

***Abstract*—The processing of moving object trajectories arises in many application domains. We focus on a trajectory similarity search, the distance threshold search, which finds all trajectories within a given distance of a query trajectory over a time interval. A multithreaded CPU implementation that makes use of an in-memory R-tree index can achieve high parallel efficiency. We propose a GPGPU implementation that avoids index-trees altogether and instead features a GPU-friendly indexing scheme. We show that our GPU implementation compares well to the CPU implementation. One interesting question is that of creating efficient query batches (so as to reduce both memory pressure and computation cost on the GPU). We design algorithms for creating such batches, and we find that using fixed-size batches is sufficient in practice. We develop an empirical response time model that can be used to pick a good batch size.**

***Keywords*-Spatiotemporal databases; distance threshold queries; GPGPU.**

I. Introduction

Applications in a wide range of domains process datasets that contain spatiotemporal trajectories (generated from, e.g., GPS devices, simulations of moving objects). We focus on historical continuous trajectories [1], where a database of trajectories is searched. More specifically, we study the *distance threshold search*: Find all trajectories within a distance d of a given query trajectory over a time interval $[t_0,t_1]$. One motivation application for this work is in the area of astrobiology [2]. The Galactic Habitable Zone is thought to be the region(s) of the Galaxy that may favor the development of complex life. Some of these regions may be inhospitable due to transient radiation events, such as supernovae explosions or close encounters with flyby stars. Studying habitability thus entails processing the following distance threshold searches on the trajectories of (possibly billions of) stars orbiting the Milky Way for various time intervals: (i) Find all stars within a distance d of a non-moving point; (ii) Find the stars that host a habitable planet and are within a distance d of all other stellar trajectories.

Numerous methods to index and process moving object trajectories efficiently have been developed in the field of spatial and spatiotemporal databases, with a focus on out-of-core implementations where the majority of the database resides on disk. Indexing techniques have been designed to optimize data layouts so as to reduced disk accesses, and most previous work focuses on sequential query processing. However, current architectures and data storage capacities offer attractive alternatives. In particular, many-core GPUs (Graphic Processing Units) should be well-suited to the large number of moving distance calculations required for processing queries on spatiotemporal databases. In addition, current compute nodes and GPU devices have relatively large memories, making it possible to use database partitioning for parallel, in-memory query processing.

In this work we focus on in-memory distance threshold searches using General Purpose Computing on Graphics Processing Units (GPGPU), which to the best of our knowledge has not been studied previously. In this context we make the following contributions:

- We propose an indexing technique for efficient distance threshold searches on the GPU. (Section IV)
- We implement an efficient GPU kernel to perform distance threshold searches. (Section V)
- We show that our GPU implementation can afford significant speedup over a previously developed CPU-only implementation. (Section VII)
- Efficient searches are predicated on grouping query trajectories in batches, and we propose several algorithms to create such batches. We find that creating same-size batches is sufficient to achieve good performance in practice. (Sections VI and VII)
- We develop an empirical response time model that can be used to select a good query batch size. (Section VIII)

Related work is discussed in Section II. The in-memory, on-GPU, distance threshold search problem is formally defined in Section III. Finally, Section IX concludes with a summary of our findings and perspectives on future work.

II. Related Work

A trajectory is defined by a set of positions that describe the motion of a moving object in Euclidean space over a time

interval. The continuous nature of trajectories requires that each point traversed be approximated by a polyline (points connected via line segments). The goal of trajectory similarity searches is to find trajectories with similar attributes. The most studied similarity search is the k Nearest Neighbors (kNN) search [3], [4], [5].

Based on the success of the R-tree [6], many index trees have been proposed to index trajectory data [7], [7], [8], [9]. Entire systems have been designed to process and analyze trajectories (e.g., TrajStore [10], SECONDO [5]). In general, the query response time is correlated to the number of index-tree node accesses, since a node access may require data to be transferred from disk to memory.

Distance threshold searches have not received a lot of attention in the literature. Our previous work in [11] studies *in-memory* sequential distance threshold searches on the CPU, using an R-tree to index trajectories inside hyperrectangular minimum bounding boxes (MBBs). We proposed an indexing method that achieves a desirable trade-off between the index overlap, the number of entries in the index, and the overhead of processing candidate trajectory segments. The work in [12] solves a similar problem, but a key difference with the work in [11], and with this work, is that they consider an *out-of-core* scenario.

Methods have been recently proposed for indexing spatial and spatiotemporal databases on GPUs [13], [14], [15], [16], some of which employ more straightforward data structures in comparison to index-trees. The efficient execution of kNN searches has been investigated on the GPU [17], [18] and on hybrid CPU-GPU environments [19], although not in the context of spatiotemporal databases. In this work, we target the GPU, but we focus on distance threshold searches.

III. Problem Definition

Let D be a database that contains 4-dimensional (3 spatial dimensions + 1 temporal dimension) *entry line segments*. Each entry line segment l_i, $i = 1, \ldots, |D|$, is defined by a spatiotemporal start extremity (x_i^{start}, y_i^{start}, z_i^{start}, t_i^{start}), a spatiotemporal end extremity (x_i^{end}, y_i^{end}, z_i^{end}, t_i^{end}), a segment id and a trajectory id. Segments belonging to the same trajectory have the same trajectory id and are ordered temporally by their segment ids. We call $t_i^{end} - t_i^{start}$ the *temporal extent* of l_i. We search the database for entry segments within a distance d of a query Q, where Q is a set of *query line segments*. The search is continuous, such that an entry segment may be within the distance threshold d of particular query segment for only a subinterval of that segment's temporal extent. For example, for a query segment with temporal extent [0,1], the search may return (l_1,[0.2,0.4]) and (l_2,[0.3,0.9]). Notions of "trajectory similarity" exist that compare trajectories at a coarser level. However, in this work we must process each trajectory segment individually to yield precise time intervals during which query segments lie within the query distance d.

We consider a platform that consists of a host, with RAM and CPUs, and a GPU device with its own RAM (global and shared) and Streaming Multi-Processors (SMPs). We consider an *in-memory database*, meaning that D is stored once and for all in global memory on the GPU. Like most previous works we focus on an *online* scenario where the objective is to minimize the response time for an arbitrary set of queries. We consider the case in which D and Q cannot fit together on the GPU for the following reasons: (i) Memory on the GPU is limited and in practice a single database is subjected to a large number of queries; (ii) Memory for the result set must be allocated statically since dynamic memory allocation is not possible on the GPU; however, the result set size is non-deterministic. Thus, memory allocation for the result set must be conservative and overestimate the amount of memory required (which grows linearly with $|Q|$), thus contributing to memory pressure on the GPU.

Our approach to reduce memory consumption on the GPU is to partition Q in batches that are processed in sequence. Note that such incremental query processing is also useful when multiple users query the database simultaneously, and would thus compete for memory space on the GPU.

IV. Trajectory Indexing

Previous works in the spatiotemporal database literature use index-trees to process queries on the CPU, including our own work, which uses an in-memory R-tree [11]. The GPU uses the Single Instruction Multiple Data (SIMD) execution model, requiring that work-items (GPU threads) that take different execution paths be executed sequentially. Index-tree traversals consists of many branch instructions, which should be avoided on the GPU [13], [14]. The authors therein question whether index-trees should be used at all on GPUs. Instead they use "flatly structured grids," data structures in which polylines are converted to MBBs and are assigned to cells on a grid to spatially partition and index the data. In this work, we design another GPU-friendly indexing scheme, which targets scenarios in which large query sets must be partitioned into batches that are processed iteratively.

In light of the above we first sort the entry segments by non-decreasing t_{start} values. Without loss of generality we assume that $t_i^{start} \leq t_{i+1}^{start}$. The full temporal extent of database D is $[t_0, t_{max}]$ where $t_0 = \min_{l_i \in D} t_i^{min}$ and $t_{max} = \max_{l_i \in D} t_i^{max}$. We divide this temporal extent logically into m bins of fixed length $b = (t_{max} - t_0)/m$. We say that an entry segment l_i, $i = 1, \ldots, |D|$, belongs to bin B_j, $j = 1, \ldots, m$, if $\lfloor t_i^{start}/b \rfloor = j$. For bin B_j we can then define $B_j^{start} = b \times j$ and $B_j^{end} = \max_{l_i \in B_j} t_i^{end}$. We call $[B_j^{start}, B_j^{end}]$ the temporal extent of bin B_j. We then define $B_j^{first} = \arg\min_{i | l_i \in B_j} t_i^{start}$ and $B_j^{last} = \arg\max_{i | l_i \in B_j} t_i^{start}$. $[B_j^{first}, B_j^{last}]$ is thus the index range of the entry segments in B_j. Bin B_j is fully described as (B_j^{start},B_j^{end}, B_j^{first}, B_j^{last}). The set of bins is the "index" of the database.

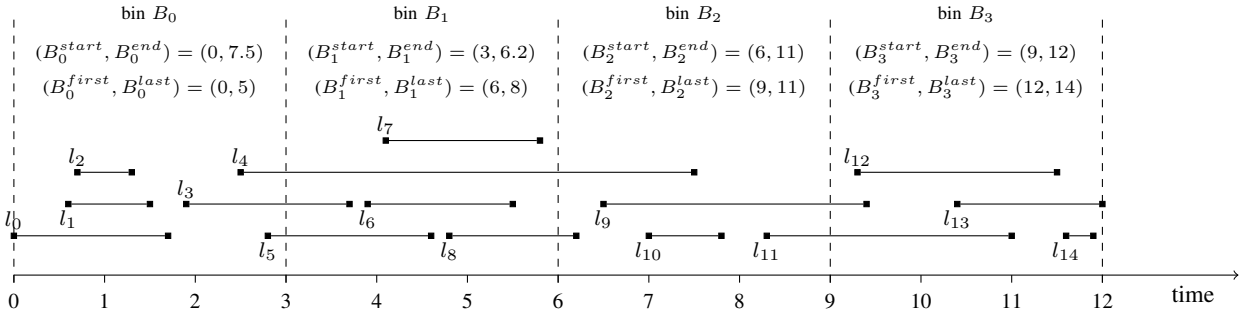


Figure 1: Example indexing of line segments into bins.

Figure 1 shows an example for a database with 14 entry segments with a total temporal extent of 12 (segments are simply shown as non-overlapping horizontal lines as we do not depict their spatial dimensions or orientations). The temporal extent of the database is logically divided into 4 bins, and for each bin we indicate the B^{start},B^{end}, B^{first} and B^{last} values. For instance, bin B_1 contains the three entry segments with t^{start} in the $[3, 6)$ interval, i.e., l_6,l_7 and l_8. Therefore, $B_1^{first} = 6$ and $B_1^{last} = 8$. Among the three entry segments in bin B_1, l_8 has the highest t^{end} value at 6.2. Therefore, $B_1^{start} = 3$ and $B_1^{end} = 6.2$.

Given the database and set of bins, we consider a query set Q. We first sort the query segments by non-decreasing t_{start} values in $O(|Q|\log|Q|)$ time, which gives the temporal extent of the query (the combined temporal extent of the query segments). We then determine the set of (contiguous) bins that temporally overlap the temporal extent of the query. We do this determination in $O(\log m)$ time by using an index-tree in which we store the bins' temporal extents. Given this set of bins, $\mathcal{B}$, we compute $first = \min_{B \in \mathcal{B}} B_j^{first}$ and $last = max_{B \in \mathcal{B}} B_j^{last}$ in $O(1)$ time. We thus obtain $E_Q = \{l_i \in D | first \leq i \leq last\}$, the set of the candidate entry segments that may be part of the result set. Each query segment must then be compared to each candidate segment in E_Q. We term each such a comparison an *interaction*, and we have a total of $|Q| \times |E_Q|$ interactions.

Some of the computed interactions are "wasteful." For instance, in the context of the example in Figure 2, consider a query with a single query segment with temporal extent [8,10]. The query segment overlaps the temporal extents of bin B_2 and bin B_3, meaning that it will be compared to $l_9, \ldots, l_{14}$. And yet, l_{10}, l_{13}, and l_{14} cannot overlap the query segment's temporal extent. More generally, the larger $|Q|$, the larger the potential number of wasteful interactions.

Figure 2 shows an example of how using batches decreases the number of interactions. The top of the figure shows the same set of bins as in Figure 1, without showing the entry segments but indicating temporal extents and numbers of entry segments. The bottom of the figure shows a set of 60 query segments partitioned into 6 same-size batches. For each query batch we indicate its temporal extent and its number of segments. An arrow is drawn between a query batch and an entry bin if the query segments in the batch must be compared to the entry segments in the bin. Below the batches we show the number of interactions necessary to process the query. For instance, batch 2 has a temporal extent (5.7,9,1), which overlaps with the temporal extents of bins B_0, B_1, and B_2, which contain 6, 3, and 3 entry segments, respectively. Therefore, the processing of batch 2 entails 10×(6+3+3)=120 interactions. Using 10-segment query batches results in a total of 450 interactions. The figure also shows the number of interactions using larger batches. For instance, while processing 10-segment batch 2 requires 120 interactions and processing 10-segment batch 3 requires 60 interactions, processing the aggregate 20-segment batch leads to 20×(6+3+3+3) = 300 $>$ 180 interactions. In this example, processing all query segments as a single 60-segment batch would lead to 900 interactions, twice the number of interactions when using 10-segment batches. Processing each query segment individually (batch size of 1) minimizes the number of segment interactions. However, processing each batch incurs the non-negligible overhead of sending data from the host to the GPU and of invoking a GPU kernel. Consequently, one of the questions we investigate in this work is that of choosing batch sizes that minimize query response time.

More advanced indexing methods could be envisioned to avoid computing wasteful interactions. However, these methods lead to more data transfer overhead between the host and the GPU as more information must be conveyed. Preliminary results show that in practice this overhead leads to significant increases in total response time in spite of reducing the number of wasteful interactions.

We propose the following general approach for implementing query distance threshold searches on the GPU. The entry segments in D, sorted by non-decreasing t^{start} values, are stored contiguously in the global memory of the GPU. The database index (the bins) and the query segments (sorted by non-decreasing t^{start} values) are stored in RAM on the host. The query segments are partitioned in batches (not

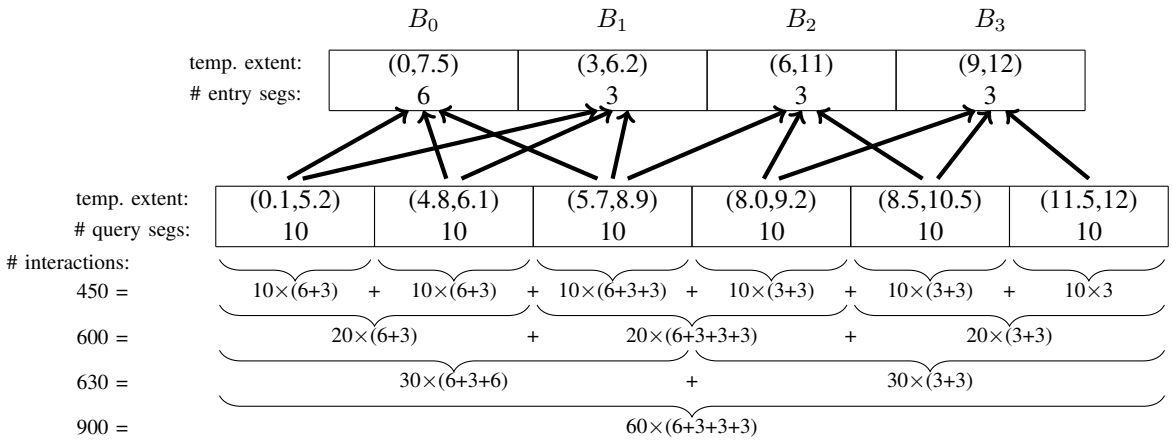


Figure 2: Example matching between query batches and entry bins.

necessarily of identical sizes). For each batch, the index range of the candidate entry segments is calculated using the bins. The query segments in a batch and the index range, encoded as two integers, are sent from the host to the GPU. The candidate entry segments are compared to the query segments, generating a result set that is returned to the CPU. Our indexing method guarantees that the candidate entry segments are stored contiguously in memory, which allows for efficient memory transfers between global, local and private memory spaces on the GPU. The search completes when all batches have been processed in this manner.

V. Search Algorithm

In this section, we describe an algorithm that performs distance threshold searches using the indexing and search techniques outlined in Section IV. This algorithm is implemented as a GPU kernel using OpenCL, and optimized to use as few branch instructions as possible. We use one GPU thread for each candidate entry segment, and each thread compares its candidate entry segment to all query segments in a query batch $Q_{batch} \subset Q$, for a total of $|Q_{batch}|$ interactions. More specifically, the kernel takes as input: (i) Q_{batch}, an array of query trajectory segments sorted by t^{start} values; (ii) *firstC*, the index in D of the first candidate entry segment; (iii) $numC$, the number of candidate entry segments; (iv) d, the threshold distance; and (v) *setID*, a global index that keeps track of the location in memory where the next result set item should be written. Q_{batch}, $firstC$, and $numC$ are computed on the host before executing the kernel and transferred to the GPU along with d and *setID*. The kernel returns a set of time intervals annotated by trajectory ids.

The pseudo-code of the kernel is shown in Algorithm 1. The threads in OpenCL are numbered using a global id ($gid \geq 0$). Threads with $gid \geq numC$ do not participate in the computation (lines 2-5). Once the result set is initialized to the empty set (line 6), the relevant candidate segment is copied into the thread's private memory (variable *entrySegment*, line 7). The algorithm then loops over all query segments to compute the interactions (line 8). Given the candidate segment and the current query segment, function *temporalIntersection()* generates new candidate and query segments that span the same time interval (line 9). The algorithm then computes the interval of time during which these two segments are within a distance d of each other (line 10), which involves computing the coefficients of and solving a degree two polynomial [5]. If this interval is non-empty, then *setID* is incremented atomically (line 12). The interval is annotated with the trajectory id and added to the result set (line 13). The full result set is returned once all interactions have been computed.

Algorithm 1 Pseudo-code for the GPU kernel.

```
1:  procedure GPUKernel(Q_batch, firstC, numC, d, setID)
2:      gid ← getGlobalId()
3:      if gid≥numC then
4:          return
5:      end if
6:      resultSet ← ∅
7:      entrySegment ← D[firstC + gid]
8:      for all querySegment ∈ Q_batch do
9:          (entrySegment, querySegment) ← temporalIntersection(
                    entrySegment,querySegment)
10:         timeInterval ← calcTimeInterval(
                    entrySegment,querySegment,d)
11:         if timeInterval ≠ ∅ then
12:             resultID ← atomic_inc(setID)
13:             resultSet[resultID] ← resultsSet[resultID] ∪ timeInterval
14:         end if
15:     end for
16:     return resultSet[0:setID]
17: end procedure
```

Although potentially as high as $|Q_{batch}|\times$*numC*, in practice the size of the result set is much smaller. Since memory for the result set must be allocated statically, in our experiments we conservatively allocate enough memory for a result set with $|D|$ entries. In practice, one could allocate much

less memory, and in the rare cases in which more memory is needed one would simply re-attempt the kernel execution with more allocated memory.

VI. Generation of Query Batches

Using small batches increases the total number of kernel invocations, and each such invocation has a non-negligible overhead. Conversely, using large batches increases the number of wasteful interactions. The temporal properties of the dataset should guide how one groups the query segments into batches. For instance, for the example in Figure 2, merging the first two sets of 10 query segments into a batch leads to no extra interactions since both sets overlap with entry bins B_0 and B_1. This is not the case when merging the third and and fourth sets of 10 query segments, as in this case 100 extra interactions are generated. While picking a good batch size is important, it is thus also important to group together query segments that together do not overlap too many entry bins. We describe hereafter several algorithms to group query segments into batches (detailed pseudo-codes are available in a companion technical report [20]).

Periodic – This algorithm uses a fixed single batch size, s, as in Figure 2. Each consecutive s queries in Q are grouped together in a batch, for a total of $|Q|/s$ batches.

SetSplit-Fixed – This algorithm takes as input a set of query segments Q and a number of batches to generate. It begins with a list of batches in which each batch contains exactly one query segment, and iteratively merges adjacent batches until the list contains the specified number of batches. At each iteration, for each possible merge of any two neighboring batches, the algorithm performs the merge that leads to the smallest increase in number of interactions. Note that this algorithm may lead to very small or very large batches, which can lead to high overhead or high numbers of wasteful interactions.

SetSplit-MinMax – To address the small/large batch problem of SetSplit-Fixed, this algorithm generates batches while imposing constraints on minimum and maximum batch sizes. It takes as input a set of query segments, Q, a lower, resp. upper, bound on the batch size, min, resp. max. In a first phase, this algorithm operates as SetSplit-Fixed, but disallowing merges that would create batches with more than max query segments. In a second phase, while there is at least one batch with fewer than min query segments, it merges this batch with its predecessor or successor batch (whichever leads to the smallest increase in number of interactions). In that phase, some batches may be generated with more than max query segments.

SetSplit-Max – This algorithm is similar to SetSplit-MinMax, but uses $min = 1$.

GreedySetSplit-Min – This algorithm takes as input a set of query segment, Q, and a lower bound on the batch size, min. It first traverses the list of batches (initially each batch contains one query segment) and performs all merges that lead to no increase in interactions. It then traverses the list of batches again, and merges each batch with fewer than min query segments with its successor.

GreedySetSplit-Max – This algorithm is similar to GreedySetSplit-Min but imposes an upper bound on batch size, max. We do not include a GreedySetSplit algorithm with both an upper and a lower bound on batch size. We did not find such an algorithm useful in practice due to the fact that the "free" merges performed in the first phase of the algorithm lead to somewhat uniform batch sizes in all our experimental scenarios.

SetSplit-Fixed and SetSplit-MinMax have $O(|Q|^2)$ complexity, while GreedySetSplit-Min and GreedySetSplit-Max have $O(|Q|)$ complexity. These four algorithms, unlike Periodic, attempt to reduce the number of interactions while avoiding high overhead.

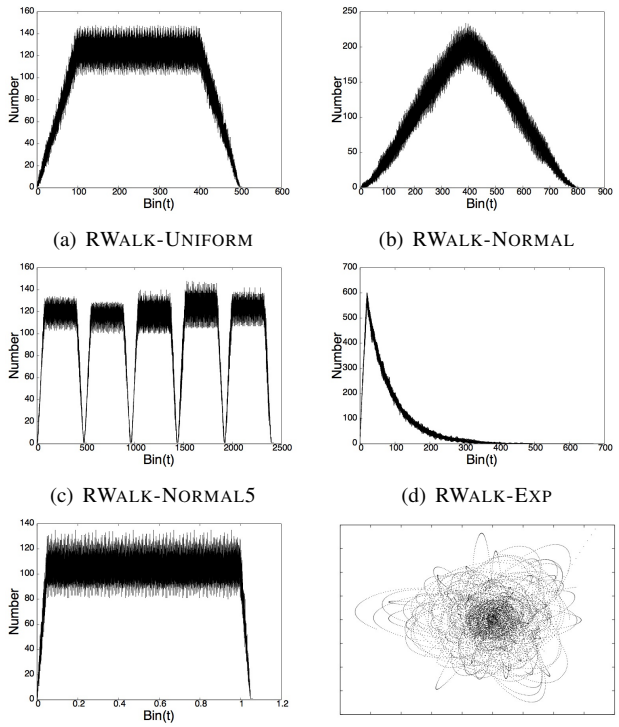
VII. Experimental Evaluation

A. Datasets

Table I: Dataset characteristics.

Dataset	Trajec.	Entries
RWalk-Uniform	2,500	997,500
RWalk-Normal	2,500	1,000,000
RWalk-Normal5	2,500	1,000,000
RWalk-Exp	10,000	684,329
Galaxy	2,500	1,000,000

We evaluate our approach using several datasets. The Galaxy dataset contains trajectories of stars moving in the Milky Way's gravitational field (as generated by the astronomy application mentioned in Section I). In this dataset the number of active trajectories throughout time is roughly uniform. Because our approach relies on temporal data partitioning, we also generate synthetic datasets with various temporal profiles. These datasets are based on trajectories of bodies subjected to Brownian motion. The RWalk-Uniform dataset consists of 400-timestep trajectories whose start times are sampled from a uniform distribution over the [0,100] interval. The RWalk-Normal dataset is similar but uses a normal distribution to generate start times, with a mean of 200 and standard deviation of 200, truncated to the [0,400] interval. The RWalk-Exp dataset consists of trajectories with numbers of timesteps that are sampled from an exponential distribution with $\lambda = 1/70$, truncated to the [2,1000] interval, with start times sampled from a uniform distribution over a the [0,20] timestep interval. The RWalk-Normal5 dataset is generated as the RWalk-Normal dataset, but one of 5 different normal distributions is randomly selected for each trajectory. This dataset thus exhibits distinct active and inactive phases. The various parameter values for generating these datasets were picked so as to produce distinct patterns of numbers of entry segments assigned to entry bins. These patterns



(a) RWALK-UNIFORM (b) RWALK-NORMAL

(c) RWALK-NORMAL5 (d) RWALK-EXP

(e) GALAXY (f) Sample GALAXY trajectories.

Figure 3: (a)-(e): number of entry segments per bin vs. bin start time for our 5 datasets. (f): sample trajectories from GALAXY projected on the x-y plane.

Table II: Experimental scenarios.

Scen.	Dataset	# Traj.	# Q Seg.
S1	GALAXY ($d = 1$)	100	40,000
S2	GALAXY ($d = 5$)	100	40,000
S3	RWALK-UNIFORM ($d = 5$)	100	39,900
S4	RWALK-UNIFORM ($d = 25$)	100	39,900
S5	RWALK-NORMAL ($d = 50$)	100	40,000
S6	RWALK-NORMAL ($d = 150$)	100	40,000
S7	RWALK-NORMAL5 ($d = 50$)	100	40,000
S8	RWALK-NORMAL5 ($d = 150$)	100	40,000
S9	RWALK-EXP ($d = 50$)	1000	52,044
S10	RWALK-EXP ($d = 100$)	1000	69,881

are shown in Figure 3(a)-(e) for each dataset. In addition, Figure 3(f) shows sample trajectories from the GALAXY dataset. Table I gives the number of trajectories and of entry segments in our datasets. These datasets are publicly available [21].

B. Experimental Methodology

The GPU-side implementation is developed in OpenCL, and the host-side implementation is developed in C++. The host-side implementation is executed on one of the 6 cores of a dedicated 3.46 GHz Intel Xeon W3690 processor with 12 MB L3 cache, while the GPU side runs on an Nvidia Tesla C2075 card. We measure query response times averaged over 3 trials (standard deviation over the trials is negligible). In all experiments the number of entry bins in our index is set to 10,000. In our experiments, we utilize the trajectory searches in Table II, where the scenario, dataset, query distance, number of query trajectories and segments are outlined. For a given set of entry segments, the response time depends on the set of query segments. However, in all scenarios we see only small response time variations across experiments with different sets of query segments (e.g., within 3.08% for 10 random trials for scenario S2). Consequently, we only present results for a single query set for each scenario.

C. Comparison to a CPU Implementation

In previous work we have developed a CPU implementation of distance threshold searches [11], [22]. This implementation uses an in-memory R-tree index, indexing entry segments into MBBs in a way that achieves a good trade-off between the number of entries in the index, the volume of the space occupied by the MBBs, and the computational cost of candidate trajectory segment processing [11]. Parallelization is straightforward using OpenMP, with high parallel efficiency (78-90%) using up to 6 cores. We have compared this CPU implementation to the GPU implementation proposed in this work. For instance, for scenario S2, and using the PERIODIC with a batch size of 120, our GPU implementation achieves an average response time of 2.08 s, compared to 6.88 s for the OpenMP CPU implementation (or a speedup of 3.3). While these results are tied to the hardware characteristics of our experimental platform, we conclude that a GPU implementation of distance threshold searches is worthwhile and can yield substantial improvement over a CPU-only version.

D. Evaluation of Batch Generation Algorithms

We now evaluate the relative merit of the algorithms for creating query segment batches (PERIODIC, SETSPLIT, GREEDYSETSPLIT). Our results correspond to a "best case" for the SETSPLIT and GREEDYSETSPLIT algorithms, for two reasons. First, the response times do not include the time necessary to compute the query batches. This time is negligible for PERIODIC, but can be significant for SETSPLIT and even for GREEDYSETSPLIT. Second, using an exhaustive search, for each experimental scenario we have determined the best parameter configuration for the algorithms (number of batches, lower/upper bounds).

Table III shows, for each algorithm and each experimental scenario, the percentage response time difference relative to the response time of the best algorithm for that experimental scenario. We show two versions of the PERIODIC algorithm. PERIODICBEST corresponds to PERIODIC when using the batch size that leads to the lowest response time for the experimental scenario at hand. PERIODICGOOD corresponds to PERIODIC but using the worst batch size in a -20/+20

neighborhood of the best batch size (i.e., the batch size in that interval that leads to the highest response time).

Over our 10 experimental scenarios, the SETSPLIT and GREEDYSETSPLIT algorithms all lead to response times that are within 3.4% of each other (SETSPLIT-FIXED leads to a significantly larger response time for S9 due to the heavy-tailed characteristic of this dataset, which causes SETSPLIT-FIXED to generate a few very large batches). The GREEDYSETSPLIT algorithms, even though they have lower complexity than the SETSPLIT algorithms, do well. GREEDYSETSPLIT-MAX, resp. GREEDYSETSPLIT-MIN, leads to the lowest response time in 2, resp. 4, of the 10 experimental scenarios. Overall, the GREEDYSETSPLIT algorithms are among the 3 best algorithms for each experimental scenario. This suggests that the quadratic complexity of the SETSPLIT algorithm to attempt a less local optimization is unnecessary.

When using the best batch size, PERIODIC can produce response time on par or even better than that of the GREEDYSETSPLIT and SETSPLIT algorithms. Overall, for each experimental scenario, PERIODICBEST leads to response times at most 1.69% larger than that of the best GREEDYSETSPLIT or SETSPLIT algorithm for that scenario. PERIODICGOOD still leads to response times at most 3.56% larger than the best GREEDYSETSPLIT or SETSPLIT algorithm over the 10 experimental scenarios.

Recall that the above response time results do not include the time to compute the batches. When adding the (CPU) time to compute the batches, the SETSPLIT algorithms lead to response times more than 4.69 times larger than PERIODICBEST on average and up to 8.84 times larger (discounting SETSPLIT-FIXED for scenario S9, which has very high response time). The GREEDYSETSPLIT algorithms fare better, with response times only up to 2.9% larger the PERIODIC over all experimental scenarios. We conclude that although computing batches that reduce wasteful interactions is appealing, in practice it does not outperform PERIODIC. In the next section, we propose a response time model for selecting the batch size used by PERIODIC.

VIII. PERFORMANCE MODELING

Because designed for GPU execution, the indexing scheme proposed in this work does not rely on index-trees. While not completely free of data-dependent behavior, the more deterministic behavior of this scheme makes it possible to predict query response time. The model consists of a GPU component and CPU component. The GPU component accounts for the invocation overhead and execution time of each individual kernel invocation, so that summing over all invocations gives the estimated GPU time for processing the entire set of query segments. The CPU component accounts for the time to perform memory allocations, set kernel parameters, send query data to the GPU, receive result sets from the GPU, marshal data, and perform other

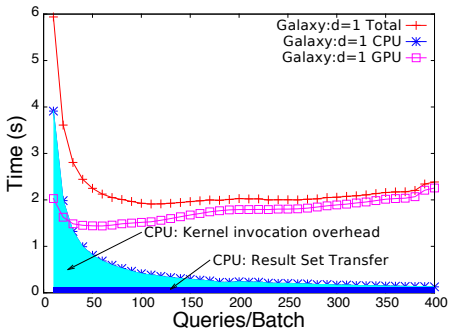


Figure 4: Response time vs. queries/batch (s) for S1 (GALAXY dataset with $d = 1$). Individual CPU and GPU components are shown.

CPU-side computations (e.g., counter and pointer updates). Figure 4 shows response time results for the S1 experimental scenario, highlighting both the CPU and GPU components. The GPU curve shows an initial decrease as the batch size, s, increases in the interval $10 \leq s \leq 40$. This is because for low s values the GPU device is underutilized and the kernel invocation overhead is large due to many such invocations. For $s \geq 50$, the GPU time increases due to the increasing number of interactions that must be computed (as explained in Section VI). The CPU time curve shows a steady decrease as s increases. This is because the smaller the value of s the more kernel invocations and thus the more work done on the CPU. We show two portions of the CPU time. The time necessary to perform kernel invocations, including the transfer of query segments from the CPU to the GPU, is shown as a shaded cyan portion below the CPU curve. The shaded blue portion corresponds to the time necessary for transferring result sets from the GPU back to the CPU.

A. GPU Component

1) Model: Let us use $T^{GPU}(i, c)$ to denote the GPU time for a kernel invocation that computes the i interactions necessary for comparing a batch of i/c query segments against c candidate segments (using c GPU threads). Let us use $\Theta^{GPU}(i, c)$ to denote the overhead of launching a no-op kernel for q query segments and i interactions (the overhead depends both on the number of queries and on the number of GPU threads). Given the i interactions to be computed, we denote by α the fraction of these interactions that lead to an item being added to the result set (i.e., both a temporal hit and a spatial hit), by β the fraction of these interactions for which the entry segment does not overlap the query segment temporally (i.e., a temporal miss), and by γ the fraction of these interactions for which the entry segment overlaps the query segment temporally but not spatially (i.e., a temporal hit but a spatial miss). We have $\alpha + \beta + \gamma = 1$. We distinguish these three cases because the computational cost is different in each. Candidate segments

Table III: Percentage response time difference relative to the lowest response time for all algorithms and experimental scenarios. Results for the algorithm with the lowest response time shown in boldface.

Algorithm	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
GREEDYSETSPLIT-MAX	0.15	0.15	0.15	**0.00**	**0.00**	0.60	1.44	0.34	1.29	0.16
GREEDYSETSPLIT-MIN	**0.00**	0.24	**0.00**	0.15	0.52	0.11	**0.00**	**0.00**	0.93	0.10
SETSPLIT-FIXED	1.11	1.69	1.03	1.02	0.92	1.03	2.34	1.52	562.90	0.62
SETSPLIT-MAX	1.50	1.97	1.02	0.35	1.51	1.54	3.37	2.78	0.90	0.69
SETSPLIT-MINMAX	0.24	0.33	0.10	0.17	0.69	**0.00**	0.77	0.93	**0.00**	**0.00**
PERIODICBEST	0.37	**0.00**	0.23	0.50	0.83	1.03	1.11	0.09	1.50	1.69
PERIODICGOOD	3.21	2.47	1.64	3.56	2.15	1.43	2.21	1.05	2.52	2.69

that are temporal misses can be determined with only a few instructions (i.e., comparing temporal extremities of query and candidate segments). Candidate segments that are temporal hits but spatial misses require more instructions (i.e., spatial extremities comparisons). Candidate segments that should be added to the result set require even more instructions to be performed (i.e., determining the actual overlapping temporal interval). One can view the computation of an interaction as a sequence of comparisons and moving distance calculations, but these comparisons and calculations are short-circuited whenever a segment is found to be a temporal or spatial miss.

We denote by $T_1(i,c)$, $T_2(i,c)$, and $T_3(i,c)$ the time for a kernel invocation with i interactions so that all c candidate segments are temporal and spatial hits, temporal misses, and temporal hits and spatial misses, respectively. This leads us to the following model:

$$T^{GPU}(i,c) = T_1^{GPU}(\alpha i, c) + T_2^{GPU}(\beta i, c) + T_3^{GPU}(\gamma i, c) - 2\Theta^{GPU}(i,c). \quad (1)$$

The first three terms above each include a $\Theta^{GPU}(i,c)$ component, hence the subtracted fourth term. $T_{GPU}(i,c)$ is computed for each batch, and the sum gives the total GPU time assuming the batch size is s:

$$T^{GPU}(s) = \sum_{j=0}^{|Q|/s} T^{GPU}(i_j, i_j/s).$$

Q is the total set of query segments (for simplicity this equation assumes that s divides $|Q|$). i_j is the number of interactions that must be computed for the j-th query batch, which is determined based on the entry segment bins (see Section IV). Therefore i_j/s is the number of candidate entry segments for the s query segments in the batch.

In this model, parameters α, β and γ depend on the dataset and the query. They must thus be determined empirically for representative scenarios. By contrast, the functions Θ^{GPU}, T_1^{GPU}, T_2^{GPU}, T_3^{GPU} depend only the hardware characteristics of the platform.

2) Estimating α, β and γ: Recall that α is, for a kernel invocation, the fraction of interactions that lead to a new item being added to the result set. Given that α is dataset dependent, we use a pragmatic approach to estimate it for a particular dataset once and for all, i.e., before the dataset is being queried in "production" use. Depending on the temporal distribution of the entry segments, there may be time periods with few active trajectories and some with many, resulting in a non-uniform distribution of query hits throughout time. To estimate α, we divide the dataset into $numEpochs$ temporal epochs. For each epoch we select a batch of s sample queries that fall within the epoch. We do this by randomly selecting s consecutive query segments from a representative query dataset. We then execute our kernel and calculate the fraction of interactions that produced result items. We perform this over enough trials such that the predicted total number of result set items is within 5% of the total true number of result set items. This procedure yields an α estimate for each epoch. This estimate may be inaccurate if the sample queries are not representative of queries that will be processed in production. Also, if too low a value of $numEpochs$ is used, then the α estimates are more likely to be inaccurate since transient temporal patterns are then averaged over larger epochs. Using $numEpochs = 1$ could be accurate for a temporally uniform dataset, but vastly inaccurate for datasets for exhibit temporal transience. In all the experiments we use $numEpochs = 50$.

Unlike α, β can be computed precisely. For a given set of s query segments, one can determine which entry segments they may temporally overlap using the bins in our indexing scheme (see Section IV). Then, with two nested loops one can simply compare the temporal extremities of each query segment to that of each entry segment, yielding an exact value for β. Parameter γ is computed as $1 - \alpha - \beta$.

To summarize, for a given dataset we compute once and for all a set of α estimates for each epoch and for the full range of (reasonable) s values. Then, for each batch of s queries we compute an α estimate, β and γ. Therefore, for each candidate s value we can plug appropriate values of these three parameters into the T^{GPU} response time model.

3) Estimating T_1^{GPU}, T_2^{GPU}, T_3^{GPU} and Θ^{GPU}: The T_1^{GPU}, T_2^{GPU}, T_3^{GPU} and Θ^{GPU} functions depend only on the implementation of the kernel and the hardware characteristics of the platform. As a result, we can empirically estimate these time components based on benchmark results. Let us consider T_1^{GPU}, i.e., the kernel response time when

all interactions are both temporal hits and spatial hits. We generate a synthetic dataset and query set in which all interactions are guaranteed to be both temporal and spatial hits. Response time results show roughly linear dependency with the number of interactions. Given a number of interactions, i, and a number of candidate entries, c, based on the benchmark results we can thus use linear interpolation to determine a response time prediction $T_1^{GPU}(i,c)$. The same approach is used to estimate $T_2^{GPU}(i,c)$, $T_3^{GPU}(i,c)$ and $\Theta^{GPU}(i,c)$.

B. CPU Component

To estimate $T_1^{CPU}(s)$, the portion of the CPU time that corresponds to the kernel invocation overhead for a batch size s (the cyan area in Figure 4), we generate a synthetic dataset with $\alpha \approx 0$. With a very low value of α, the result set has negligible size. As a result, kernel response time is approximately equal to the aggregate kernel invocation overhead. We thus obtain a kernel invocation overhead curve for the full range of batch sizes. This benchmark must be executed for various total numbers of query segments so that, for a query set Q, our model uses benchmark results obtained for approximately $|Q|$ query segments. For our platform we obtain (the R^2 value for the fit is above 0.9999):

$$T_1^{CPU}(s) = -0.0017 + 32.2946 \times s^{-0.9528}. \quad (2)$$

To estimate T_2^{CPU}, the portion of the CPU time that corresponds to the transfer of the result set from the GPU to the CPU (the blue area in Figure 4), we rely on the α parameter defined in the GPU component of our response time model and estimated as described in Section VIII-A2. Using α, we can estimate the number of result set items generated by each kernel invocation. Summing over all kernel invocations and multiplying by the size in bytes of a result set item yields the total size of the result set, which we denote by σ. Assuming that T_2^{CPU} does not depend on s, we can estimate it by dividing σ by the GPU-CPU bandwidth measured on the platform. For our platform we obtain:

$$T_2^{CPU}(\sigma) = 1.54 \times 10^{-8} \times \sigma. \quad (3)$$

In the end, the total response time is modeled as $T^{GPU}(s) + T_1^{CPU}(s) + T_2^{CPU}(s,\sigma)$.

C. Model Evaluation

Figure 5 shows actual and modeled response times vs. the batch size for a subset of our experimental scenarios. The CPU and GPU model components are shown with separate curves. The general trends of the actual response time are respected by the model. The results shown in Figure 5 are representative of the results for all our experimental scenarios, which are provided in a companion technical report [20]. In some instances the model tracks the actual response time well, while in others it exhibits some deviations.

Table IV: Model-based batch size vs. Best batch size.

Search	Model	Actual	Slowdown
S1	80	110	4.80%
S2	80	120	6.30%
S3	80	120	4.50%
S4	80	110	4.50%
S5	100	140	2.80%
S6	100	140	2.30%
S7	140	160	0.80%
S8	150	160	0.58%
S9	170	220	0.10%
S10	200	210	0.97%

The main purpose of our model is to produce a sufficiently coherent prediction so that a good batch size can be selected. Table IV summarizes results for our 10 experimental scenarios, where the Model column gives the model-based batch size, the Actual column gives the empirically best batch size, and the Slowdown column gives the response time slowdown due to using the model-driven batch size, as a percentage.

The slowdown is at most 6.3% and negligible in several cases. We conclude that, in spite of data dependency challenges, our model is useful for determining a good batch size to use with the PERIODIC algorithm and for predicting the total query response time. An interesting question is whether our modeling approach can be used for response time prediction purposes for other spatiotemporal queries.

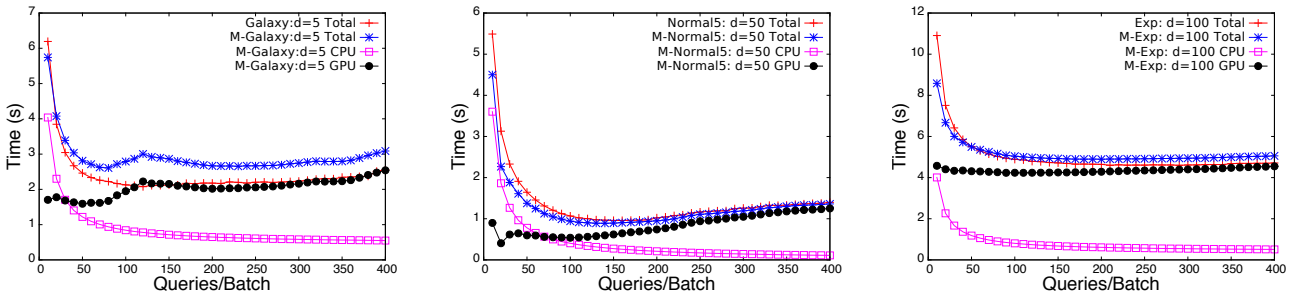
IX. CONCLUSIONS

We have studied the execution of in-memory distance threshold searches on the GPU. We have shown that the parallelism afforded by the GPU, provided a GPU-friendly indexing method is used, can outperform a multithreaded CPU implementation that uses an in-memory R-tree index. We have proposed several algorithms for partitioning a query set into individual batches, so as to reduce memory pressure and computational cost on the GPU. We have found that a simple algorithm that partitions the query set into fixed-size batches leads to competitive response times. We have developed an empirical response time model that makes it possible to pick a good batch size. This is in spite of the data-dependent behavior of the spatiotemporal database searches and in part because our indexing method is more deterministic than index-tree traversals.

A short-term future directions is to extend our performance model to the case when multiple workqueues are used to overlap computation with communication between the CPU and the GPU. A broader direction is to apply our indexing method and our response time model to other types of spatiotemporal queries on the GPU.

ACKNOWLEDGMENTS

This material is based on work supported by the National Aeronautics and Space Administration through the NASA



(a) S2: GALAXY. (b) S7: RWALK-NORMAL5. (c) S10: RWALK-EXP.

Figure 5: Modeled response times vs. queries per batch (s) for searches S2, S7, S10. The red curve shows the actual response time, the blue curve shows the modeled total response time, where the CPU (magenta) and GPU (black) model components added together equal the modeled total (blue) curve.

Astrobiology Institute under Cooperative Agreement No. NNA08DA77A issued through the Office of Space Science.

REFERENCES

[1] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, "A data model and data structures for moving objects databases," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2000, pp. 319–330.

[2] M. G. Gowanlock, D. R. Patton, and S. M. McConnell, "A Model of Habitability Within the Milky Way Galaxy," *Astrobiology*, vol. 11, pp. 855–873, 2011.

[3] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Algorithms for Nearest Neighbor Search on Moving Object Trajectories," *Geoinformatica*, vol. 11, no. 2, pp. 159–193, 2007.

[4] Y.-J. Gao, C. Li, G.-C. Chen, L. Chen, X.-T. Jiang, and C. Chen, "Efficient k-nearest-neighbor search algorthims for historical moving object trajectories," *J. Comput. Sci. Technol.*, vol. 22, no. 2, pp. 232–244, 2007.

[5] R. H. Güting, T. Behr, and J. Xu, "Efficient k-nearest neighbor search on moving object trajectories," *The VLDB Journal*, vol. 19, no. 5, pp. 687–714, 2010.

[6] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1984, pp. 47–57.

[7] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Proc. for Moving Object Trajectories," in *Proc. of the 26th Intl. Conf. on Very Large Data Bases*, 2000, pp. 395–406.

[8] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications," in *Proc. of the Intl. Conf. on Multimedia Computing and Systems*, 1996, pp. 441–448.

[9] V. P. Chakka, A. Everspaugh, and J. M. Patel, "Indexing large trajectory data sets with seti," in *Proc. of the Conf. on Innovative Data Sys. Research*, 2003, pp. 164–175.

[10] P. Cudre-Mauroux, E. Wu, and S. Madden, "TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets," in *Proc. of the 26th Intl. Conf. on Data Engineering*, 2010, pp. 109–120.

[11] M. Gowanlock and H. Casanova, "In-Memory Distance Threshold Queries on Moving Object Trajectories," in *Proc. of the Sixth Intl. Conf. on Advances in Databases, Knowledge, and Data Applications*, 2014, pp. 41–50.

[12] S. Arumugam and C. Jermaine, "Closest-Point-of-Approach Join for Moving Object Histories," in *Proc. of the 22nd Intl. Conf. on Data Engineering*, 2006, pp. 86–95.

[13] J. Zhang, S. You, and L. Gruenwald, "Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs." *Information Systems*, vol. 44, pp. 134–154, 2014.

[14] ——, "U^{2}STRA: High-performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs," in *Proc. of the 2012 ACM Workshop on City Data Management Workshop*, ser. CDMW '12, 2012, pp. 5–12.

[15] S. You, J. Zhang, and L. Gruenwald, "Parallel spatial query processing on gpus using r-trees," in *Proc. of the 2nd ACM SIGSPATIAL Intl. Workshop on Analytics for Big Geospatial Data*, ser. BigSpatial '13. New York, NY, USA: ACM, 2013, pp. 23–31.

[16] L. Luo, M. D. F. Wong, and L. Leong, "Parallel implementation of r-trees on the gpu," in *Design Automation Conf. (ASP-DAC), 2012 17th Asia and South Pacific*, Jan 2012, pp. 353–358.

[17] J. Pan and D. Manocha, "Fast GPU-based Locality Sensitive Hashing for K-nearest Neighbor Computation," in *Proc. of the 19th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems*, 2011, pp. 211–220.

[18] K. Kato and T. Hosino, "Multi-gpu algorithm for k-nearest neighbor problem," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 1, pp. 45–53, 2012.

[19] M. Kruliš, T. Skopal, J. Lokoč, and C. Beecks, "Combining CPU and GPU architectures for fast similarity search," *Distributed and Parallel Databases*, vol. 30, no. 3–4, pp. 179–207, 2012.

[20] Gowanlock, M. and Casanova, H., "Parallel Distance Threshold Query Processing for Spatiotemporal Trajectory Databases on the GPU," http://arxiv.org/abs/1405.7461, May 2014, Technical Report.

[21] http://navet.ics.hawaii.edu/%7Emike/datasets/GPU/trajdat.zip, accessed 25-May-2014.

[22] M. Gowanlock, H. Casanova, and D. Schanzenbach, "Parallel In-Memory Distance Threshold Queries on Trajectory Databases," in *Proc. of the Sixth Intl. Conf. on Advances in Databases, Knowledge, and Data Applications*, 2014, pp. 80–83.