Indexing of Spatiotemporal Trajectories for Efficient Distance Threshold Similarity Searches on the GPU

Michael Gowanlock Henri Casanova
Information and Computer Sciences Department
University of Hawai'i at Mānoa, Honolulu, HI, U.S.A.
[*gowanloc,henric*]*@hawaii.edu*

***Abstract*—Applications in many domains search moving object trajectory databases. The distance threshold search finds all trajectories within a given distance of a query trajectory. We develop three GPU distance threshold search implementations that use indexing techniques significantly different from those used in CPU implementations. We determine experimentally under which conditions each approach performs well using one real-world astrophysics dataset and two synthetic datasets. Overall, we find that the GPU is an attractive technology for a broad range of relevant trajectory database scenarios.**

***Keywords*-Distance threshold similarity search; GPGPU; moving object databases; query optimization.**

I. Introduction

Trajectory data is generated in many application domains, e.g., from global positioning systems (GPS) devices, from scientific simulations, and from geographical information systems (GIS). In this work, we focus on historical continuous searches [1], where an input trajectory database is searched to gain domain-specific insight. In particular, we study the *distance threshold search*: Find all trajectories within some distance of a given query trajectory over a time interval. This work is motivated by an astrophysics/astrobiology application [2]. Astrobiology is the study the origin, evolution, distribution and future of life in the universe. The Milky Way hosts many rocky, low-mass planets that may be capable of supporting complex life. However, some regions of the Milky Way may be inhospitable due to transient radiation events (such as supernovae) or close encounters with flyby stars that can gravitationally perturb planetary systems. Studying habitability thus entails solving the following two types of *distance threshold searches* on stellar trajectories: (i) Find the stars that host a habitable planet and are within a distance d of a supernova explosion; and (ii) Find the stars that host a habitable planet and are within a distance d of any other stellar trajectory. In both cases, the time intervals in which these events occur are obtained.

Most previous work on spatial and spatiotemporal databases focuses on sequential searching of on-disk databases. In this work, instead, we focus on parallel searching of in-memory databases. Due to increased available memory and to the proliferation of multi-core and manycore architectures, parallel in-memory implementations can provide significant performance improvements over sequential out-of-core implementations. Furthermore, spatiotemporal trajectory datasets can trivially be partitioned and queried in-memory across multiple hosts in parallel, e.g., in a cluster with GPU-equipped compute nodes.

With this rationale, and because distance threshold searches require large numbers of polyline comparisons, we explore their processing using General Purpose Computing on Graphics Processing Units (GPGPU) making the following contributions. We develop three indexing schemes and accompanying search algorithms for efficient distance threshold searches on the GPU. We evaluate these implementations and a CPU-only implementation with a real-world astrophysics dataset and two synthetic datasets.

II. Related Work

A spatiotemporal trajectory is the set of the positions of an object over time, connected by polylines (line segments). A goal in the spatiotemporal database community is to perform *trajectory similarity searches*, i.e., finding trajectories within a database that exhibit similarity in terms of spatial and/or temporal proximity or features [3], [4], [5], [6]. A trajectory similarity search used in many application areas is the kNN (k Nearest Neighbors) search [7], [8], [9].

Similarity searches proceed in two phases: (i) search an index to obtain a candidate result set; (ii) use refinement to produce the final result set. Several index-trees, whose traversals are pruned to minimize disk accesses, have been proposed and implemented in systems such as TrajStore [10] and SECONDO [9]. Index-trees have been used extensively for kNN searches. Unfortunately, pruning index tree traversals is not possible for distance threshold searches (because k is unknown) [11].

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 index [12]. The work in [13] solves a similar problem but part of

the database resides on disk. Indexing methods for in-memory moving object trajectory databases have been advanced for the GPU [14], [15], [16], [17]. Some indexes for the GPU are less sophisticated than index-trees because branch instructions used for the tree traversals cause thread serialization and thus poor parallel efficiency [18]. Previous work has studied kNN searches (but not on trajectories) for the GPU [19], [20] and for hybrid CPU-GPU environments [21]. In this work we focus on distance threshold searches for the GPU, which to our knowledge has only been explored in our previous work [22]. That previous work assumes that the query set cannot fit entirely in GPU memory, thereby requiring back-and-forth communication between the host and the GPU. In this work, we assume that the query set fits on the GPU, which makes it possible to explore a different range of indexing schemes (while still considering several memory constraints).

III. Problem Statement

Each trajectory is defined by a series of line segments. In all that follows, we refer to these trajectory line segments, instead of individual trajectories. Let D be a spatiotemporal database that contains n 4-dimensional (1 temporal and 3 spatial dimensions) *entry line segments*. A segment l_i, $i = 1, \ldots, |D|$, is defined by a spatiotemporal start point (x_i^{start}, y_i^{start}, z_i^{start}, t_i^{start}), an end point (x_i^{end}, y_i^{end}, z_i^{end}, t_i^{end}), a segment id and a trajectory id. We call $t_i^{end} - t_i^{start}$ the *temporal extent* of l_i. The distance threshold search searches for entry segments within a distance d of a query set Q, where Q is a set of *query line segments*, $q_k, k = 1, \ldots, |Q|$. The search is continuous, i.e., the result set contains query/entry pairs annotated by the time interval during which the two segments are within a distance d of each other. For example, a search may return (q_1,l_1,[0.1,0.3]) and (q_1,l_2,[0.5,0.95]), for a query segment q_1 with temporal extent [0,1].

We consider a host, with RAM and CPUs, and a GPU with its own memory and Streaming Multi-Processors connected to the CPU via a (PCI Express) bus. We assume an *in-memory database*, meaning that D is stored once and for all in global memory on the GPU. The objective is to minimize the response time for processing the query set Q, which is initially stored on the host but can fit entirely in GPU memory. Our intended use case is when D is partitioned across multiple GPU-equipped compute nodes in a cluster so that aggregate GPU memory is large. These assumptions, however, do not obviate all memory management issues. The *candidate result set* (i.e., the entry segments returned by the index because they may overlap the query segment) and the final result set have non-deterministic sizes that depend on the spatiotemporal characteristics of D and Q. Since there is no true dynamic memory allocation on the GPU, one must statically allocate buffers and handle buffer overflow. Memory pressure is exacerbated because on the GPU hundreds of concurrent threads produce candidate entry segments. Note that these issues are typically not problematic on the CPU (larger memory, dynamic memory allocations, fewer threads).

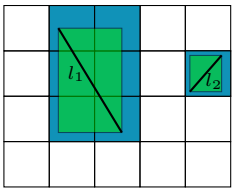

Figure 1: 2-D example rasterization of two entry segment MBBs (green) to grid cells (blue) in a 4×5 FSG.

IV. Indexing and Searching Trajectories

In this section, we propose three distance threshold search approaches for the GPU. Our implementations use OpenCL, but in this paper, we use the more common CUDA terminology (GPU vs. device, kernel vs. program, thread vs. work-item, etc.). To ensure good load balancing, all three approaches assign one query segment to each GPU thread. Assuming that $|Q|$ is moderately large then all GPU cores can be utilized.

A. Spatial Indexing

Previous work has proposed flatly structured grids (FSGs) to index 2-D spatial trajectory data on the GPU [15]. An interesting question is whether FSGs are effective even when the data has a temporal dimension. In what follows, we describe GPUSpatial, an FSG-based approach for spatiotemporal distance threshold searches on the GPU.

1) Trajectory Indexing: A FSG is a 3-D rectangular box partitioned into 3-D cells. Each entry segment $l_i \in D$ is contained in a spatial Minimum Bounding Box (MBB), and assigned to the FSG by rasterizing this MBB to grid cells. Figure 1 shows a 2-D example. We store the FSG as an array of *non-empty* cells, G. Each cell is denoted as C_h, $h = 1, \ldots, |G|$, where h is a linear coordinate computed from the cell's x, y, and z coordinates using row-major order. Cell C_h is described by an index range $[A_h^{min}, A_h^{max}]$ in an integer "lookup" array, A. If l_i's MBB overlaps C_h, then $i \in \{A[A_h^{min}], \ldots, A[A_h^{max}]\}$. Since l_i's MBB can overlap multiple grid cells, i can occur multiple times in array A. This scheme is depicted in Figure 2. Its main objective is to reduce the memory footprint of the index. We only index non-empty grid cells, and do not store cell spatial coordinates but instead recompute them from h whenever needed. Also, because we use

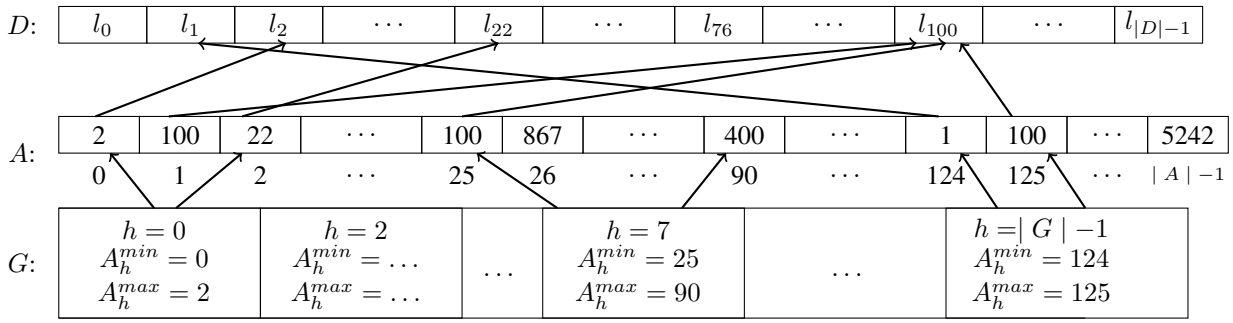

Figure 2: GPUSPATIAL example: grid array (G), lookup array (A) and database of entry segments (D).

an indirection via A, G consists of same-size elements (as opposed to picking an element size large enough to accommodate the cell with the largest number of entry segments, thereby wasting memory space). D, A, and G are stored in GPU memory before the search begins.

2) Search Algorithm: We do not sort the query segments in Q by any spatial or temporal dimension. Temporal sorting would not make sense for a spatial index, and sorting by a single spatial dimension is not effective for 3-D data. For each query segment q_k the kernel first calculates the FSG cells that overlap q_k's MBB. For each such cell, the kernel uses a binary search to find in $O(\log|Q|)$ time whether the cell is in array G. For each such cell, the indices of the entry segments it contains are obtained as $A[A_h^{min}], \ldots, A[A_h^{max}]$ and appended to a buffer U_k. With a spatial index there is no good way to store entry segments contiguously (sorting by a single spatial dimensions does not guarantee contiguous accesses). This is why we resort to using buffer U_k instead of, e.g., a 2-integer index range in a contiguous array of entry segments. Each entry in U_k is then compared to q_k to see if it is within the threshold distance. While the segments are expected to be spatially close (given their FSG overlap), they may not overlap temporally. For the example in Figure 2 consider a query q_1 (not shown in the figure), which overlaps grid cells C_0, C_1, and C_7. Cell C_1 is not in G, i.e., there is no element for $h = 1$ in G, thus C_1 does not contain any entry segments. Therefore, the only two cells to consider are C_0 and C_7, which have $[A^h_{min}, A^h_{max}]$ values of [0,2] and [25,90], respectively. In lookup array A, we find that [0,2] corresponds to entries 2, 100, and 22, while [25,90] corresponds to entries 100, 867, ..., 400. These indices are copied from A into buffer U_k. The algorithm does not remove duplicate indices (like index 100 in this example) and thus may perform redundant segment comparisons.

Since the number of entry segments that overlap q_k's MBB can be arbitrarily large, the use of buffer U_k creates memory pressure. We define an overall buffer size, s, that is split equally among all queries ($|U_k| = s/|Q|$). If the capacity of U_k is exceeded then the thread terminates and the host re-invokes the kernel with the remaining unprocessed query segments. For each re-invocation memory pressure is lower because fewer queries are executed (i.e., $|U_k|$ is larger).

Algorithm 1 GPUSPATIAL kernel.

```
1:  procedure SEARCH(G, A, D, Q, queryIDs, U, d, redo, resultSet)
2:      gid ← getGlobalId()
3:      if queryIDs = ∅ and gid≥|Q| return
4:      if queryIDs ≠ ∅ and gid≥|queryIDs| return
5:      if queryIDs = ∅ then
6:          queryID ← gid
7:      else
8:          queryID ← queryIDs[gid]
9:      (overflow , candidateSet) ← getCandidates(G, A, D,
    Q[queryID], U, d)
10:     if overflow then
11:         atomic: redo ← redo ∪ { queryID }
12:         return
13:     for all entryID ∈ candidateSet do
14:         result ← compare(D[entryID],Q[queryID])
15:         if result ≠ ∅ then
16:             atomic: resultSet ← resultSet ∪ result
17:     return
```

The pseudo-code of the search algorithm is shown in Algorithm 1. It takes the following arguments: (i) the FSG array (G); (ii) the lookup array (A); (iii) the database (D); (iv) the set of queries (Q); (v) an array that contains the ids of the queries to be reprocessed (*queryIDs*), which is empty for the first kernel invocation; (vi) buffer space (U); (vii) the query distance (d); (viii) an output array in which the kernel stores the ids of the queries that must be reprocessed (*redo*); and (ix) the memory space to store the result set (*resultSet*). Arguments that lead to array transfers betwen the host and the GPU, either as input or output, are shown in boldface. Other arguments are either pointers to preallocated zones of (global) GPU memory or integers. The algorithm begins by checking the global thread id and aborts if it is greater than Q or |queryIDs|,

depending on whether this is a first invocation or a re-invocation (lines 3-4). The id of the query assigned to the GPU thread is then acquired from Q or using an indirection via *queryIDs* (lines 5-8). Function *getCandidates* searches the FSG and returns a boolean that indicates whether buffer space was exceeded and the (possibly empty) set of candidate entry segment ids (line 9). If buffer space was exceeded, then the query id is atomically added to the *redo* array and the thread terminates (line 10-12). The algorithm then loops over all candidate entry segment ids (line 13), compares each entry segment spatially and temporally to the query (line 14) and atomically adds a query result, if any, to the result set (line 16). Once all GPU threads have completed, *resultSet* and *redo* are transferred back to the host. If |*redo*| is non-zero, then the kernel is re-invoked, passing *redo* as *queryIDs*. Duplicates in the result set are filtered out on the host.

B. Temporal Indexing

In this section, we propose a purely temporal partitioning strategy, GPUTEMPORAL. The indexing scheme is similar to that used in our previous work [22] but the search algorithm is different due to the different memory constraint assumptions.

1) Trajectory Indexing: We sort the entries in D by ascending t_{start} values. The temporal extent of D is $[t_{min}, t_{max}]$ where $t_{min} = \min_{l_i \in D} t_i^{start}$ and $t_{max} = \max_{l_i \in D} t_i^{end}$. We partition D's temporal extent into m logical bins of fixed length $b = (t_{max} - t_{min})/m$. We assign each entry segment l_i to bin B_j if $\lfloor t_i^{start}/b \rfloor = j$. There can be temporal overlap between the entry segments in adjacent bins. For each bin B_j we defined its start times as $B_j^{start} = j \times b$ and its end time as $B_j^{end} = \max((j+1) \times b, \max_{l_i \in B_j} t_i^{end})$. The temporal extent of bin B_j is defined as $[B_j^{start}, B_j^{end}]$, and $D = \bigcup_j [B_j^{start}, B_j^{end}]$. We define $B_j^{first} = \arg\min_{i | l_i \in B_j} t_i^{start}$ and $B_j^{last} = \arg\max_{i | l_i \in B_j} t_i^{start}$, i.e., the ids of the first and last entry segments in bin B_j, respectively. $[B_j^{first}, B_j^{last}]$ forms the index range of the entry segments in B_j. Each bin B_j is thus fully described as (B_j^{start},B_j^{end}, B_j^{first}, B_j^{last}) and together the bins form the database index.

2) Search Algorithm: Before performing the search, query segments in Q are sorted by non-decreasing t_{start} values, in $O(|Q| \log |Q|)$ time. For each query segment q_k, we then calculate the index range of the contiguous bins that it overlaps temporally. In practice, there are many temporally contiguous query segments, so the search can be done incrementally in near-constant time. Let $\mathcal{B}_k$ denote the set of contiguous bins that temporally overlap q_k. In constant time we can compute the index range of the candidate entry segments that may overlap q_k: $E_k = [\min_{B \in \mathcal{B}_k} B_j^{first}, \max_{B \in \mathcal{B}_k} B_j^{last}]$. We term the mapping between q_k and E_k a *schedule*, which we denote by S. Each GPU thread compares a single query q_k to the segments in D whose indices are in the E_k range. In our implementation all the above computation is performed on the host because doing it on the GPU yielded no performance gain. This is because the aforementioned near-constant time incremental search would require arbitrary thread synchronization and communication, which cannot be performed across thread blocks. Regardless, computing S on the CPU represents a negligible portion of the overall response time.

Algorithm 2 GPUTEMPORAL kernel.

```
1:  procedure SEARCH(D, Q, S, d, resultSet)
2:      gid ← getGlobalId()
3:      if gid≥|Q| return
4:      queryID ← gid
5:      entryMin ← S[gid].EntryMin
6:      entryMax ← S[gid].EntryMax
7:      for all entryID ∈ {entryMin,...,entryMax} do
8:          result ← compare(D[entryID],Q[queryID])
9:          if result ≠ ∅ then
10:             atomic: resultSet ← resultSet ∪ result
11:     return
```

The pseudo-code of the search algorithm is shown in Algorithm 2. It takes the following arguments: (i) the database (D); (ii) the query set (Q); (iii) the schedule (S); (iv) the query distance (d); and (v) the memory space to store the result set (*resultSet*). As in Algorithm 1, arguments that lead to array transfers between the host and the GPU are shown in boldface. The algorithm first checks the global thread id and aborts if it is greater than $|Q|$ (line 3). The query assigned to the thread is then acquired from Q (line 4). Next, the algorithm retrieves the minimum and maximum entry segment indices from the schedule (lines 5-6). From line 7 to 11 the algorithm operates as Algorithm 1.

C. Spatiotemporal Indexing

Expectedly, a purely temporal indexing scheme has poor temporal selectivity while a purely spatial scheme has poor spatial selectivity. Poor selectivity leads to wasteful comparisons of query and entry segments. In this section, we propose a spatiotemporal scheme, GPUSPATIOTEMPORAL, that aims for both spatial and temporal selectivity without the drawbacks of GPUSPATIAL (multiple indirections, buffer space needed).

1) Trajectory Indexing: GPUSPATIOTEMPORAL uses a temporal index but subdivides each temporal bin into spatial subbins to achieve both temporal and spatial selectivity. Entry segments are assigned to m temporal bins exactly as in GPUTEMPORAL. We compute the minimum and maximum spatial coordinates over all entry segments in D in each dimension (e.g., $x_{min} = \min_{l_i \in D}(\min(x_{start}^i, x_{end}^i))$

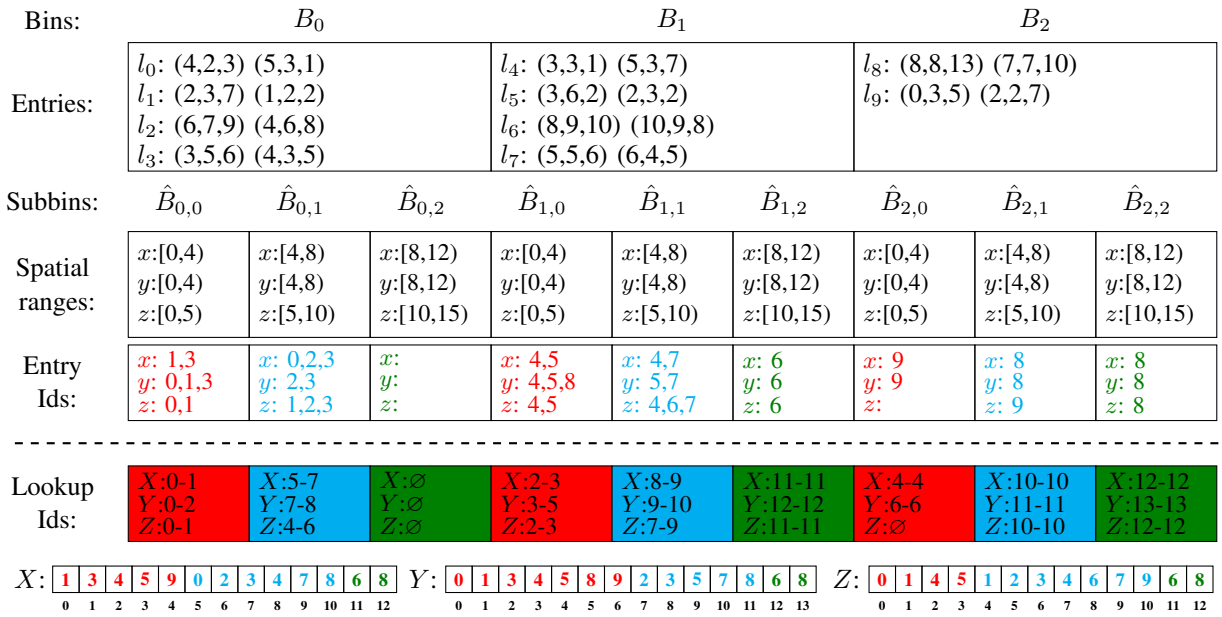

Figure 3: GPUSPATIOTEMPORAL indexing example. X, Y and Z arrays are shown at the bottom.

and $x_{max} = \max_{l_i \in D}(\max(x^i_{start}, x^i_{end}))$ for the x dimension). We then compute the maximum spatial extent of the entry segments in each dimension (e.g., $\max_{l_i \in D}|x^i_{start} - x^i_{end}|$ for the x dimension). For each temporal bin B_i, we create v spatial subbins $\hat{B}_{i,j}$, $j = 1, \ldots, v$, with the constraint that these subbins are larger than the maximum spatial extent of the entry segments. For instance, in the x dimension, this constraint is $v \leq (x_{max} - x_{min})/\max_{l_i \in D}|x^i_{start} - x^i_{end}|$. The rationale for this constraint is explained in the description of the search algorithm hereafter.

Figure 3 shows an example for 10 entry segments. The part of the figure above the dashed line shows how entry segments are logically assigned to bins and subbins. The very top of the figure shows $m = 3$ temporal bins, B_0 to B_2. Each temporal bin contains contains the segments with ids in the range $[B_j^{first}, B_j^{last}]$. For instance, $B_1^{first} = 4$ and $B_1^{last} = 7$. Each entry segment is described by its id and 2 spatial (x, y, z) extremities. For instance, segment l_6 is in temporal bin B_1 and its spatial extremities are $(8, 9, 10)$ and $(10, 9, 8)$. For clarity, in this figure we do not show any information pertaining to time (i.e., line segment and bin temporal extents). The top of the figure simply shows in which temporal bin each line segment falls. Below the temporal bins, we depict 9 spatial subbins, $\hat{B}_{0,0}$ to $\hat{B}_{2,2}$, with indicated spatial ranges in the x, y, and z dimension. For each subbin and each dimension, we show the overlapping entry segment ids. For instance, subbin $\hat{B}_{0,1}$ is overlapped in the x dimension by l_0, l_2 and l_3, in the y dimension by l_3, and in the z dimension by l_1 and l_3. The part of the figure below the dashed line shows how the logical assignment of segments to spatial subbins is implemented physically in memory. We create three integer arrays, X, Y, and Z, depicted at the bottom of the figure. Each array stores the ids of the entry segments that overlap the subbins in one spatial dimension. The ids for a subbin are stored contiguously, for the subbins $\hat{B}_{i,j}$'s sorted by (j, i) lexicographical order. This is illustrated using colors in the figure and amounts to storing contiguously all ids in the first subbins of the temporal bins, then all ids in the second subbins of the temporal bins, etc. For instance, for the y dimension, array Y consists of $v = 3$ chunks. The first chunk corresponds to the ids in subbins $\hat{B}_{0,0}$ (l_0,l_1,l_3), $\hat{B}_{1,0}$ (l_4, l_5, l_8), and $\hat{B}_{2,0}$ (l_9), the second chunk corresponds to the ids in subbins $\hat{B}_{0,1}$ (l_2, l_3), $\hat{B}_{1,1}$ (l_5, l_7), and $\hat{B}_{2,1}$ (l_8), and the third chunk corresponds to the ids in subbins $\hat{B}_{0,2}$ (none), $\hat{B}_{1,2}$ (l_6), and $\hat{B}_{2,2}$ (l_8). The reason for storing the ids in this manner is because, given our constraints on spatial subbin sizes, most queries will not overlap multiple subbins in all three dimensions. Identifying potential overlapping entry segments then amounts to examining the i-th subbin of contiguous temporal bins, for some $0 \leq i \leq v$. For the example in Figure 3, this means examining a sequence of same-color subbins.

Given the X, Y, and Z array, each spatial subbin is then described with the index range of the entries in those arrays, i.e., 6 integers. In our example, $\hat{B}_{0,1}$'s description is index range $5 - 7$ in the x dimension

(i.e., it overlaps with segments $l_{X[5]}$ to $l_{X[7]}$ in the x dimension), index range $7-8$ in the y dimension (i.e., it overlaps with segments $l_{Y[7]}$ and $l_{Y[8]}$ in the Y dimension), and index range $4-6$ in the z dimension (i.e., it overlaps with segment $l_{Z[4]}$, $l_{Z[5]}$ and $l_{Z[6]}$ in the z dimension). This indirection allows fixed-size encoding of each spatial subbin. When compared to GPUTEMPORAL, GPUSPATIOTEMPORAL only requires additional memory for the X, Y, and Z integer arrays, i.e., $\gtrsim 3|D|\times 4$ bytes.

2) Search Algorithm: As in GPUTEMPORAL we first sort Q on the host and calculate the temporally overlapping entries from the temporal bins. We also compute the set of spatially overlapping subbins in each dimension. Computing the exact set of entry segments that belong to these subbins turns out to be inefficient because we would then have to send a list of entry segment indices to the GPU. Instead, we opt for sending only a fixed set of indices at the cost of poorer spatial selectivity. We pick the spatial dimension in which the number of entry segments that overlap the query segment is the smallest. We then simply send an index range, 2 integers, in the X, Y, or Z array, depending on the dimension that was chosen. Experiments show that the induced wasteful computation on the GPU (i.e., comparisons of query and entry segments that do not overlap in one of the other two spatial dimensions) is worth the savings in amount of data sent to the GPU. This approach exploits the structure of the X, Y, and Z arrays. For the example in Figure 3, consider a query segment that overlaps temporal bins 0 and 1, and overlaps spatially with subbins $\hat{B}_{0,1}$ and $\hat{B}_{1,1}$ in the x dimension (entries 0,2,3,4,7), with subbins $\hat{B}_{0,1}$ and $\hat{B}_{1,1}$ in the y dimension (entries 2,3,5,7), and with subbins $\hat{B}_{0,1}$ and $\hat{B}_{1,1}$ in the z dimension (entries 1,2,3,4,6,7). Because only 4 entries are overlapped in the y dimension we compare the queries with those entries. The entry indices are stored *contiguously* in array Y, at indices 7 to 10. So we simply compare the query to the entry segments stored in array Y from index 7 to index 10, which is encoded in constant space. We perform a wasteful comparison with entry segment 5 due to our non-perfect spatial selectivity of entry segments.

On the host, we generate a schedule S, which contains for each query segment q_k a specification of which lookup array to use (0 for X, 1 for Y, or 2 for Z) and an index range into that array, which we encode using 4 integers (which preserves alignment). GPUSPATIOTEMPORAL requires only 1 extra indirection in comparison to GPUTEMPORAL, and avoids storing the overlapping entry indices in a buffer like in GPUSPATIAL. We sort S based on the lookup array specification so as to reduce thread divergence. Calculating S on the host takes negligible time. Recall that we enforce a minimum size for the spatial subbins. This is for two reasons. First, with small subbins each entry segment would overlap many subbins with high probability. As a result, the query id could occur many times in arrays X, Y, and/or Z, thereby wasting memory and causing redundant calculations. Second, a query that overlaps multiple subbins along all three spatial dimensions may lead to duplicates in the result set. To avoid duplicates, we default to the GPUTEMPORAL scheme whenever duplicates would occur, which wastes computation due to loss of spatial selectivity (but hopefully occurs with low probability given the relatively large subbin sizes).

Algorithm 3 GPUSPATIOTEMPORAL kernel.

```
1:  procedure SEARCH(X,Y,Z,D,Q,S,d, resultSet)
2:      gid ← getGlobalId()
3:      if gid≥|Q| return
4:      queryID ← gid
5:      arraySelector ← {X, Y, Z}
6:      if S[gid].arrayXYZ ≠ -1 then
7:          arrayXYZ ← arraySelector[S[gid].arrayXYZ]
8:          entryMin ← S[gid].entryMin
9:          entryMax ← S[gid].entryMax
10:         for all i ∈ {entryMin, ..., entryMax} do
11:             entryID = arrayXYZ[i]
12:             result ← compare(D[entryID],Q[queryID])
13:             if result ≠ ∅ then
14:                 atomic: resultSet ← resultSet ∪ result
15:     else
16:         Lines 5-10 in Algorithm 2.
17:     return
```

The pseudo-code of the search algorithm is shown in Algorithm 3. It takes the following arguments: (i) the X, Y, and Z arrays; (ii) the database (D); (iii) the query set (Q); (iv) the schedule (S); (v) the query distance (d); and (vi) the memory space to store the result set (*resultSet*). As in Algorithm 2, arguments that lead to array transfers between the host and the GPU are shown in boldface. The algorithm begins by checking the global thread id and aborts if it is greater than $|Q|$ (line 3). The query assigned to the thread is acquired from Q (line 4). A helper array is constructed that holds pointers to the X, Y, and Z arrays (line 5). If schedule S does not give a specification for one of the X, Y, or Z arrays (S[gid].arrayXYZ = -1) then the algorithm defaults to the temporal scheme (line 15). Otherwise, the algorithm retrieves the pointer to the correct X, Y, or Z array (line 7) and determines the index range for the entry segments (lines 8-9). It then processes the entry segments (lines 10-14) as Algorithm 2.

V. EXPERIMENTAL EVALUATION

A. Datasets

We evaluate the performance of our distance threshold searches using 3 spatiotemporal datasets:

Random – A small, sparse synthetic dataset that consists of 2,500 trajectories generated via random walks over 400 timesteps, for a total of 997,500 entry segments. Trajectory start times are sampled from a uniform distribution over the [0,100] interval.

Merger – A large real-world dataset[1] from the field of astrophysics that consists of particle trajectories that simulate the merger of the disks of two galaxies. It contains the positions of 131,072 particles over 193 timesteps for a total of 25,165,824 entry segments.

Random-dense – A denser synthetic dataset that is motivated by astronomy applications and generated as follows. Consider the stellar number density of the solar neighborhood, i.e., at galactocentric radius $R_\odot = 8$ kpc (kiloparsecs), of Reid et al. [24], $n_\odot = 0.112$ stars/pc^3. *Random-dense* has the same number of particles as in one disk in the *Merger* dataset (65,536) and 193 timesteps, yielding 12,582,912 entry segments, but matching the density of [24] (thus requiring a cubic volume of $65536/0.112 = 585142$ pc^3). Trajectories are generated as random walks as for *Random*. This dataset aims to represent a density consistent within the range of possible stellar densities within the Milky Way.

B. Experimental Methodology

Our implementations are in OpenCL on the GPU side and in C++ (using gcc with level 3 optimization) on the host side. The GPU-side implementation runs on an Nvidia Tesla C2075 card with 6GiB of RAM and 448 cores. The host-side implementation is executed on one of the 6 cores of a dedicated 3.46 GHz Intel Xeon W3690 processor with 12 MiB L3 cache. In all experiments we measure response time as an average over 3 trials (standard deviation is negligible). We allocate a buffer to hold the result set of the search on the GPU that can hold 5.0×10^7 items. The response time does not include the time to build the index and to store D and the index in GPU memory, which is done off-line before the search begins.

We consider three experimental scenarios, each for one of our datasets and for a range of query distances:

- S1: The *Random* dataset and a query with 100 trajectories each with 400 timesteps for a total of 39,900 query segments.
- S2: The *Merger* dataset and a query set with 265 trajectories each with 193 timesteps for a total of 50,880 query segments.
- S3: The *Random-dense* dataset and a query set with 265 trajectories each with 193 timesteps for a total of 50,880 query segments.

We also evaluate CPU-RTREE, a CPU-only implementation developed in our previous work [11], [25]. CPU-RTREE uses an in-memory R-tree index [12] and is multithreaded using OpenMP. Threads traverse the R-tree in parallel, each for a different query segment, and returns candidate entry segments. All executions of CPU-RTREE use 6 threads on our 6-core CPU. Results in [22] show that this implementation achieves around 80% parallel efficiency on 6 cores. Like for our GPU implementations, response time measurements do not include the time to build the index. One important driver of response time for index trees is how trajectory segments are assigned to MBBs [26], [27], [11]. CPU-RTREE stores $r \geq 1$ segments per MBB, with larger r implying lower index search time but more candidate entry segments to process. For all experiments presented hereafter we executed CPU-RTREE with a range of values for r, and only report on results for the r value that leads to the lowest response time. See [28] for comprehensive result graphs.

[1]This dataset was obtained from Josh Barnes [23].

C. Results for the Random Dataset

We used GPUSPATIAL with a range of FSG resolutions (i.e., numbers of grid cells). Using too coarse a resolution increases response time due to poor spatial selectivity, which leads to large numbers of segment comparisons and to many GPU threads overflowing their buffers (U_k) thus requiring multiple kernel invocations. Using too fine a resolution also increases response time because entry segments overlap many cells thus producing duplicates in the result set. Although filtering out these duplicates on the CPU takes negligible time, transferring them back to the host incurs non-negligible overhead. In our experiments, we have found that using 50 grid cells per dimension leads to the lowest response time. Regardless of the FSG resolution, we observe rapid response time increases as d increases. The disposition of the FSG index to prefer small d values is mentioned in [15]. This suggests that FSGs may not be particularly useful for spatiotemporal trajectory searches unless query distances are small.

GPUTEMPORAL's response time does not depend on d because entry segments are selected based on their temporal extents. We have used GPUTEMPORAL with various numbers of temporal bins. With too few temporal bins, poor temporal selectivity leads to large numbers of segment comparisons. But as the number of bins increases the response time reaches a minimum. In these experiments, using more than 10,000 bins does not improve temporal selectivity.

When using GPUSPATIOTEMPORAL, with 10,000 temporal bins, we find that for lower d values greater numbers of spatial subbins are desirable. This is because it is unlikely that a query will overlap multiple subbins, which would cause our algorithm to default to GPUTEMPORAL. As d increases, queries overlap

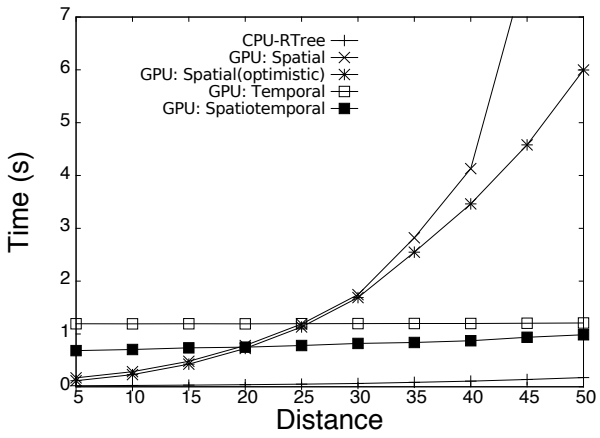

Figure 4: Response time vs. d for our 4 implementations for S1. For GPUSPATIAL we also plot an optimistic curve that ignores kernel re-invocation overheads.

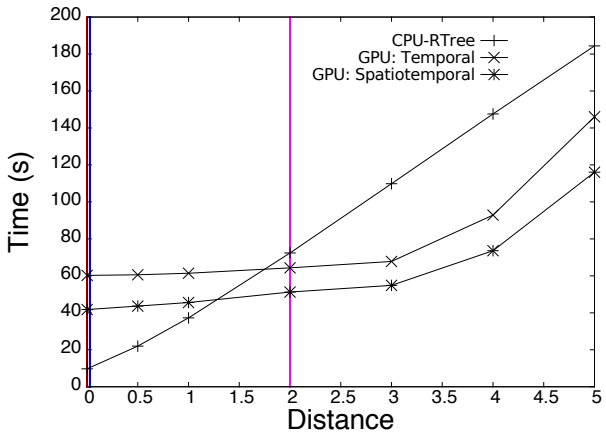

Figure 5: Response time vs. d for our implementations for S2. We indicate three distance thresholds relevant to the study of the habitability of the Milky Way. Red: close encounters between stars and planetary systems [29]; Blue: supernova events on habitable planetary systems [2], and Magenta: studying the effects of gamma ray bursts on habitable planets [30]. Both the red and blue lines are close to the vertical axis.

multiple spatial subbins with higher probability and better performance is thus achieved with fewer subbins. Overall, picking $v = 4$ subbins leads to low response time across all query distances even though for some query distances using up to $v = 16$ can further reduce response time. Comparing results between GPUSPATIOTEMPORAL with 1 subbin and GPUTEMPORAL allows us to quantify the effect of the additional indirection in GPUSPATIOTEMPORAL. At $d = 50$ (yielding the greatest number of indirections), the response time of GPUTEMPORAL is 1.21 s and that of GPUSPATIOTEMPORAL with 1 subbin is 1.36 s, or a 12.4% increase in response time due to the extra indirection.

Figure 4 shows response time vs. d for our four implementations. Each implementation is configured with parameter values based on previous results in this section (GPUSPATIAL: 50 grid cells per dimension, GPUTEMPORAL: 10,000 temporal bins, GPUSPATIOTEMPORAL: 10,000 temporal bins and 4 spatial subbins). CPU-RTREE is best across all query distances. GPUSPATIAL performs better than GPUTEMPORAL and GPUSPATIOTEMPORAL when $d < 20$, but does not scale well as d increases. This lack of scalability is not solely due to kernel re-invocation overhead. Figure 4 plots an "optimistic" curve that discounts this overhead and shows the same trend. GPUTEMPORAL and GPUSPATIOTEMPORAL have consistent response times across query distances, with an advantage for GPUSPATIOTEMPORAL. We conclude that for small and sparse trajectory datasets the overhead of using the GPU is simply too great.

D. Results for the Merger Dataset

On the larger *Merger* dataset, GPUSPATIAL leads to extremely high response times. For some values of d, GPUSPATIAL and GPUSPATIOTEMPORAL process Q incrementally due to memory space constraints for the result set at the cost of increased response time. Results for GPUTEMPORAL are similar to those for the *Random* dataset (using 1,000 temporal bins leads to the lowest response time consistently across all d values). For GPUSPATIOTEMPORAL, also using 1,000 temporal bins, we find that using $v = 16$ subbins leads to the best results for most d values. Picking a good v value can thus likely be done for a large dataset regardless of the queries and the query distance.

Figure 5 compares the CPU-RTREE, GPUTEMPORAL (with 1,000 temporal bins), and GPUSPATIOTEMPORAL (with 1,000 temporal bins and $v = 16$ subbins) and shows d values relevant to our application domain. GPUSPATIAL is omitted. GPUSPATIOTEMPORAL outperforms GPUTEMPORAL across the board by at least 23.6%. CPU-RTREE is best at low query distances but is overtaken by GPUSPATIOTEMPORAL at $d \sim 1.5$. At $d = 0.001$ the response time for CPU-RTREE is 9.70 s vs. 41.75 s for GPUSPATIOTEMPORAL (the GPU is 330.4% slower). At $d = 5$ these response times become 184.4 s and 116.09 s respectively (or the GPU is 58.8% faster). We conclude that GPUSPATIOTEMPORAL outperforms CPU-RTREE when using large datasets and large query distances.

E. Results for the Random-dense Dataset

As for the *Merger* dataset, GPUSPATIAL leads to extremely high response times and GPUTEMPORAL

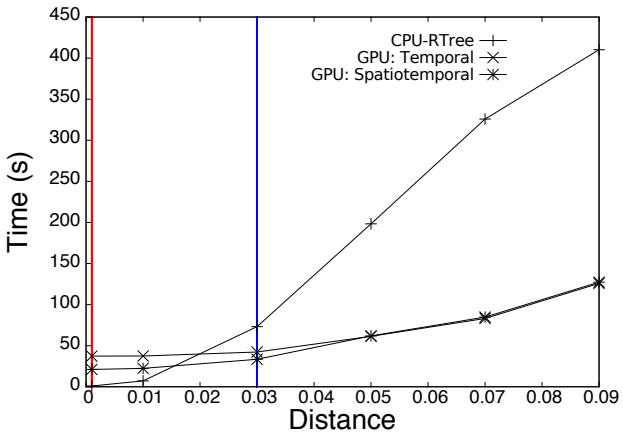

Figure 6: Response time vs. d for CPU-RTREE, GPUTEMPORAL, and GPUSPATIOTEMPORAL for S3.

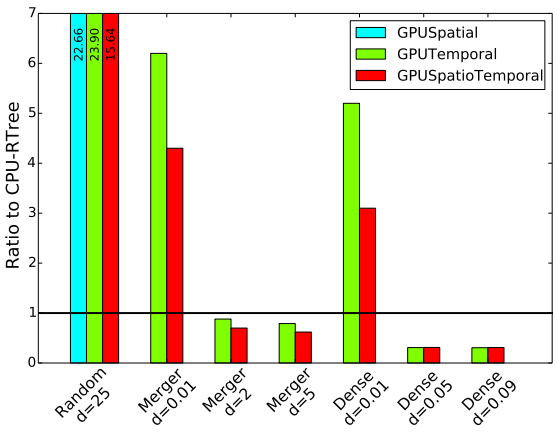

Figure 7: Ratio of GPU to CPU response times for various datasets and query distances. Values below the $y = 1$ line indicate improvements over CPU-RTREE.

using 1,000 temporal bins leads low response time consistently across all query distances. Results for GPUSPATIOTEMPORAL show that the use of subbins is only effective for low query distances. Because the dataset is denser, queries are more likely to fall within multiple subbins, in which case GPUSPATIOTEMPORAL defaults to GPUTEMPORAL. For instance for $d = 0.03$ and $v = 2$ subbins, GPUSPATIOTEMPORAL defaults to GPUTEMPORAL for almost 40% of the queries. When $v = 4$, then GPUSPATIOTEMPORAL always defaults to GPUTEMPORAL. Given the density of *Random-dense*, for larger values of d only a fraction of the queries can be solved per kernel invocation as there is insufficient memory space for the result set. Since *Random-dense* has half as many entries as *Merger*, we can increase the size of the buffer on the GPU for the result set (from 5×10^7 elements for *Merger* to 9.2×10^7 elements for *Random-dense*). This buffer size increase leads to decreases in response time due to fewer host-GPU communications. For instance, at $d = 0.09$ (which requires the greatest number of kernel invocations), the increased buffer size reduces response time by 65.76%.

Figure 6 shows response time vs. d for S3, for CPU-RTREE, GPUTEMPORAL (1,000 temporal bins), and GPUSPATIOTEMPORAL (1,000 temporal bins, $v = 4$ subbins) with the larger buffer sizes for a range of query distances. CPU-RTREE is best for $d \lesssim 0.02$ but is outperformed by the GPU implementations for larger d. At $d = 0.05$, the GPU implementations are 223% faster than CPU-RTREE. Comparing Figures 5 (*Merger* dataset) and 6 (*Random-dense* dataset) shows that the range of query distances for which using the GPU outperforms the CPU is much larger for the denser dataset. (Consider the vertical lines that correspond to relevant application scenarios.) In the astrophysics domain, datasets denser than *Random-dense* are common (i.e., to study the galactic regions at $R < 8$ kpc).

VI. CONCLUSIONS

In this paper, we have proposed indexing methods and accompanying algorithms for efficient distance threshold similarity searches on spatiotemporal trajectory datasets. Our main result is that GPU-friendly indexing methods can outperform a CPU implementation that uses an in-memory R-tree index. Figure 7 shows the ratio of the response times of the GPU implementations to the CPU implementation for our 3 datasets for selected query distances. The main findings are that although the CPU is preferable for small and sparse datasets, the GPU leads to significant improvements for large and/or dense datasets unless query distances are small. When there are large query distances, or a dense dataset is considered, the parallelism afforded by the GPU is beneficial and the overhead of using the GPU is a small fraction of the total response time. However, when the dataset is sparse and/or the query distance is small, this overhead precludes performance gains when using the GPU. Large and dense datasets are routine in some applications, including our driving application domain (astrophysics). Overall, a spatiotemporal indexing method that achieves both temporal and spatial selectivity but *without* resorting to an index tree, is effective on the GPU. The future trends for GPU technology (faster host–GPU bandwidth, increased memory, etc.) will be a further advantage for GPU implementations of spatiotemporal similarity searches.

The main future direction for this work is to apply our indexing techniques to other spatial/spatiotemporal trajectory searches and investigating hybrid implementations of the distance threshold search that uses the CPU and the GPU concurrently.

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] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, "Trajectory Pattern Mining," in *Proc. of the 13th ACM Intl. Conf. on Knowledge Discovery and Data Mining*, 2007, pp. 330–339.

[4] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, "Discovery of Convoys in Trajectory Databases," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1068–1080, 2008.

[5] M. R. Vieira, P. Bakalov, and V. J. Tsotras, "On-line discovery of flock patterns in spatio-temporal data," in *Proc. of the 17th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Inf. Syst.*, 2009, pp. 286–295.

[6] Z. Li, M. Ji, J.-G. Lee, L.-A. Tang, Y. Yu, J. Han, and R. Kays, "MoveMine: Mining Moving Object Databases," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2010, pp. 1203–1206.

[7] 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.

[8] 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.

[9] 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.

[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] 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.

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

[14] 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, no. 0, pp. 134–154, 2014.

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

[16] 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*, 2013, pp. 23–31.

[17] J. Kim, S. Kim, and B. Nam, "Parallel multi-dimensional range query processing with R-trees on GPU," *J. Parallel Distrib. Comput.*, vol. 73, no. 8, pp. 1195–1207, 2013.

[18] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 3:1–3:8.

[19] 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 Inf. Syst.*, 2011, pp. 211–220.

[20] K. Kato and T. Hosino, "Multi-GPU algorithm for k-nearest neighbor problem," *CCPE*, vol. 24, no. 1, pp. 45–53, 2012.

[21] 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.

[22] M. Gowanlock and H. Casanova, "Distance Threshold Similarity Searches on Spatiotemporal Trajectories using GPGPU," in *Proc. of the 21st IEEE Intl. Conf. on High Performance Computing*, 2014.

[23] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[24] I. N. Reid, J. E. Gizis, and S. L. Hawley, "The Palomar/MSU Nearby Star Spectroscopic Survey. IV. The Luminosity Function in the Solar Neighborhood and M Dwarf Kinematics," *Astronomical Journal*, vol. 124, pp. 2721–2738, 2002.

[25] 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.

[26] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos, "Efficient indexing of spatiotemporal objects," in *Proc. of the 8th Intl. Conf. on Extending Database Technology: Advances in Database Technology*, 2002, pp. 251–268.

[27] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento, "A trajectory splitting model for efficient spatio-temporal indexing," in *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, 2005, pp. 934–945.

[28] M. Gowanlock and H. Casanova, "Towards Efficient Indexing of Spatiotemporal Trajectories on the GPU for Distance Threshold Similarity Searches," http://arxiv.org/abs/1410.2698, October 2014, Technical Report.

[29] J. J. Jiménez-Torres, B. Pichardo, G. Lake, and A. Segura, "Habitability in Different Milky Way Stellar Environments: A Stellar Interaction Dynamical Approach," *Astrobiology*, vol. 13, pp. 491–509, 2013.

[30] B. C. Thomas, A. L. Melott, C. H. Jackman, C. M. Laird, M. V. Medvedev, R. S. Stolarski, N. Gehrels, J. K. Cannizzo, D. P. Hogan, and L. M. Ejzak, "Gamma-Ray Bursts and the Earth: Exploration of Atmospheric, Biological, Climatic, and Biogeochemical Effects," *Astrophysical Journal*, vol. 634, pp. 509–533, 2005.