Distance Threshold Similarity Searches: Efficient Trajectory Indexing on the GPU

Michael Gowanlock*† Henri Casanova*

* Information and Computer Sciences Department

University of Hawai'i at Mānoa, Honolulu, HI, U.S.A.

henric@hawaii.edu

† Massachusetts Institute of Technology, Haystack Observatory

Westford, MA, U.S.A.

gowanloc@mit.edu

Abstract

Applications in many domains perform searches over datasets that contain moving object trajectories. A common class of searches are similarity searches that attempt to identify trajectories with similar characteristics. In this work, we focus on the distance threshold similarity search that finds all trajectories within a given distance of a query trajectory over a time interval. This search involves large numbers of Euclidean moving distance calculations, thus making it a good candidate for execution on manycore platforms such as GPUs. However, low search response time is preconditioned on efficient indexing of trajectory data. We propose three indexing schemes designed for the GPU, with spatial, temporal and spatiotemporal selectivity. These schemes differ significantly from traditional tree-based indexing schemes that have been previously proposed for CPU executions. We evaluate implementations of our proposed indexing schemes using two synthetic and one real-world astrophysics dataset, showing under which conditions each scheme achieves high performance. Our broad finding is that a GPU implementation, provided an appropriate indexing scheme is used, can outperform a multithreaded CPU implementation that uses a state-of-the-art index tree. In particular, the performance improvement is large for regimes that are relevant for classes of real-world applications, thereby demonstrating that the GPU is an attractive platform for searching and processing moving object trajectories.

1 INTRODUCTION

Trajectory data is generated in a wide range of application domains, such as the motions of people or objects captured by global positioning systems (GPS), the movement of objects in scientific applications, such as stars in astrophysical simulations, vehicles in traffic studies, animals in zoological studies and a range of applications of geographical information systems (GIS). We study historical continuous trajectories [1], where a trajectory dataset is given as input and is searched to gain domain-specific insight. A broad class of searches are *trajectory similarity searches*, i.e., searches that find trajectories that have similar spatial and/or temporal features (proximity, shape, clustering behavior, etc.). In this work we study a particular similarity search, the *distance threshold search*: Find all trajectories within a distance d of a given query trajectory over a given time interval [2].

Searching a trajectory dataset for those objects that are within a threshold Euclidean distance of each other is a natural idea. The initial motivation for this work comes from the astrobiology domain [3]. Astrobiology is the study the evolution, distribution and future of life in the universe. The past decade of exoplanet searches implies that the Milky Way, and hence the universe, hosts many rocky, low mass planets that may be capable of supporting complex life. Some regions of the Milky Way may be inhospitable due to transient radiation events, such as supernovae explosions or close encounters with flyby stars that can gravitationally perturb planetary systems. Studying habitability thus entails solving the following distance threshold searches on the trajectories of stars orbiting in the Milky Way: (i) Find all stars within a distance d of a supernova explosion (or gamma ray burst), i.e., a non-moving point over a time interval; and (ii) Find the stars, and corresponding time periods, that host a habitable planet and are within a distance d of all other stellar trajectories.

The spatial and spatiotemporal database communities have developed efficient trajectory indexing and processing methods. Many of these methods focus on sequential implementations, where a fraction of the data is stored in memory and the rest is stored on disk. Minimizing disk accesses is thus the main objective. Alternatively, with relatively large memories available in modern workstations, sizable in-memory databases have become feasible. Furthermore, with the proliferation of multicore and manycore architectures, parallel in-memory implementations can provide significant performance improvements over sequential out-of-core implementations. In instances where memory capacity on a single node is insufficient, historical continuous trajectory datasets can be partitioned and queried independently in-memory across multiple compute nodes.

For these reasons, we focus on in-memory trajectory database on a single node.

Distance threshold searches require large numbers of Euclidean moving distance calculations so as to determine precise spatiotemporal object proximity. As a result, attractive platforms for executing these searches are manycore GPUs whose SIMD (Single Instruction Multiple Data) execution model should allow for large numbers of concurrent distance calculations. Regardless of the execution model, a key technique to reduce database search response time is indexing. Several indexing schemes for spatiotemporal trajectory data have been proposed for out-of-core databases and used for in-memory databases for searches processed on the CPU. These techniques typically rely on index trees and are not necessarily appropriate on GPU architectures.

In this work, we focus on enabling efficient distance threshold searches on spatiotemporal trajectory databases on the GPU. We develop three GPU-friendly indexing schemes (spatial, temporal, spatiotemporal) suitable for distances threshold searches on the GPU, and develop a GPU kernel for each scheme. We then compare our GPU implementations to a previous CPU-only implementation that uses an in-memory R-tree index, and show that using the GPU can afford significant speedup. Interestingly, we find that for large datasets efficient trajectory splitting strategies for an R-tree index, at least for the in-memory case, provides limited or no performance improvements. We then evaluate our implementations with 4-D datasets (3 spatial dimensions and 1 temporal dimension), including a real-world astrophysics dataset (of a galaxy merger) and two synthetic datasets. Our key finding is that distance threshold searches on the GPU can have response times significantly lower than that on the CPU when trajectory datasets are large, dense, and/or the distance threshold is large.

The paper is outlined as follows. Section 2 discusses related work. Section 3 formally defines the problem. Section 4 describes our indexing techniques and search algorithms. Section 5 presents our experimental results. Finally, Section 6 concludes with a summary of our findings and a discussion of future research directions.

2 Background and Related Work

A key question in database research is the efficient retrieval of data. While database management systems can support arbitrary queries on arbitrary data, more efficient retrieval can be achieved in specific domains if there is structure on the data and/or if particular queries are expected. Such a domain is that of spatiotemporal databases that store the trajectories of moving objects. A trajectory is a collection of points connected by

polylines (i.e., a set of line segments). A typical goal of trajectory databases is to perform *trajectory similarity searches*, i.e., finding trajectories that exhibit similarity in terms of spatial and/or temporal proximity, or exhibit similarity in terms of spatial and/or temporal features. Similarity searches have been studied in various domains, such as convoys [4], flocks [5], and swarms [6]. A well-known trajectory similarity search is the kNN (k Nearest Neighbors) search [7], [8], [9], [10].

The typical similarity search approach proceeds in two phases: (i) search an index to obtain a candidate set; (ii) use refinement to produce the final result set. The search phase focuses on *pruning*, i.e., avoiding traversing parts of the index. To this end, several index-trees have been proposed as inspired by the success of the popular R-tree [11], such as TB-trees [12], STR-trees [12], 3DR-trees [13], SETI [14], and implemented in systems such as TrajStore [15] and SECONDO [10]. More specifically, these works map nodes in an index-tree to pages stored on disk. The goal is to minimize the number of accessed index-tree nodes so as to avoid costly data transfers between memory and disk. Index-trees have been used extensively for kNN searches.

In this work we study distance threshold searches, which can be viewed as kNN searches with an unknown value of k and thus unknown result set size. As a result several of the aforementioned index-trees are not efficient as the index search cannot be pruned. Distance threshold searches, although relevant to several application domains, have not received a lot of attention in the literature. Our previous work in [16] studies in-memory sequential distance threshold searches, using an R-tree to index trajectories inside hyperrectangular minimum bounding boxes (MBBs). The main contribution therein is 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 [17] solves a similar problem, but assume that part of the database resides on disk. Other trajectory similarity searches rely on metrics of similarity at coarse grained resolutions [18]. Instead, the distance threshold search requires precise comparisons between individual polylines. The large number of such comparisons is the main motivation for using the GPU.

In the context of in-memory moving object trajectory databases, several authors have explored the use of multicore and manycore architectures. Spatial and spatiotemporal indexing methods have been advanced for the GPU [19], [20], [21], [22], [23], [24]. Given the single instruction multiple data (SIMD) nature of the GPU, proposed indexes for this architecture tend to be less sophisticated than the index-trees used for out-of-core

databases. This is in part because branches in the instruction flow cause thread serialization and thus loss of parallel efficiency [25]. The kNN query (not on trajectories) has been studied in the context of the GPU [26], [27] and on hybrid CPU-GPU environments [28]. In this work we focus on indexing techniques for distance threshold similarity searches on trajectories for the GPU, which to our knowledge has only been explored in our previous work [29]. That previous work assumes that the query set cannot fit entirely on the GPU due to memory constraints, thereby requiring back-and-forth communication between the host and GPU. Instead, in this work we assume the query set fits on the GPU, which mandates different indexing schemes.

3 Problem Statement

Distance threshold search on the GPU – Let D be a spatiotemporal database of n 4-dimensional (3 spatial and 1 temporal dimensions) *entry line segments*. A line 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. 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. The distance threshold search searches for entry segments within a distance d of a query set Q, where Q is a set of line segments that belong to a series of moving object trajectories. We call the line segments in Q *query segments* and denote them by $q_k, k = 1, \ldots, |Q|$. The search is continuous, such that an entry segment may be within the distance threshold d of a particular query segment for only a subinterval of that segment's temporal extent. We call a comparison between an entry segment and a query segment an *interaction*. The result set thus contains a set of query and entry segment pairs, and for each pair 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 the above search on a platform that consists of a host, with RAM and CPUs, and a GPU with its own memory and Streaming Multi-Processors (SMPs) connected to the CPU via a (PCI Express) bus. We consider an *in-memory database*, meaning that D is stored once and for all in global memory on the GPU, i.e., the database is stored once and queried multiple times. The objective is to minimize the response time for processing the queries in Q. This is the typical objective considered in other spatiotemporal database works such as the ones reviewed in Section 2. We consider the case in which both D and Q can fit in GPU memory.

This means that GPU memory is large enough and not shared with other users. Our intended scenario is that of a distributed memory environment in which multiple GPUs (e.g., within compute nodes and across compute nodes) are reserved by a user. As explained in Section 1, D can be partitioned and Q can be replicated across GPUs, so as to enable in-memory distance threshold searches for databases larger than the physical memory of a single GPU. To ensure good load balancing, all of our approaches assign one query segment to each GPU thread. Assuming that $|Q|$ is moderately large, then all GPU cores can be utilized.

Moving distance calculation – The distance threshold search performs spatiotemporal comparisons between entry and query segments. Each comparison amounts to a moving distance computation, which requires a large number of floating point operations (see Appendix B in [30] for details). The algorithms described in upcoming sections have GPU threads invoke a `compare()` function to perform these comparisons. Threads can take three possible execution paths in this function due to three distinct scenarios: (i) the two line segments do not overlap temporally; (ii) the two line segments overlap temporally but are not within spatial distance d of each other; or (iii) the two segments overlap temporally and are within spatial distance d of each other (a query hit). As a result, GPU threads invoking this function concurrently will experience branch divergence [25] since some threads will return from `compare()` earlier than others. Note that this is different from typical divergent scenarios in which all threads end at the same point but take different execution paths. Those threads that take a longer execution path (query hits) perform significant amounts of SIMD computation and achieve high parallel efficiency. Since the search is data-dependent, it is not possible to determine ahead of time which execution path will be taken for a given query, and therefore not possible to avoid branch divergence.

Memory management – Our distance threshold search proceeds in the two typical steps (see Section 2). In the first step, an index is searched so as to determine a set of candidate line segments that may be part of the final result set. The final result set cannot be returned directly because the index is constructed based on line segment MBBs. As a result, due to "wasted space" in the index [16], [31], the MBB of a line segment and that of a query segment may overlap, but the line segment does not necessarily fall within the threshold distance of the query segment. In a second step, each candidate segment is compared to the query segment and potentially added to the result set. The number of candidate segments and the number of segments in the result set are data-dependent and cannot be determined before the search executes. In CPU implementations of in-

memory distance threshold searches [16], [17], [32], memory for holding the candidate sets and the final result sets is either allocated/deallocated dynamically or pre-allocated conservatively (overestimating the memory requirement by a factor $|Q|$). On the CPU these memory management issues are not problematic in practice since the number of threads is limited (e.g., set to the number of physical cores) and the memory is large and can be easily dynamically allocated. By contrast, on the GPU these issues are problematic, even though we assume that both D and Q fit in memory. First, there is a large number of threads that each need memory to store candidate segments concurrently, leading to memory pressure [23]. Second, on the GPU an upper bound on the memory requirement must be defined before execution of the search (there is no true dynamic memory allocation). As a result, one must define a fixed size for a statically allocated memory buffer for each thread. If the memory requirements exceed this buffer then one must perform a series of kernel invocations so as to "batch" the generation of the candidate sets and the final result set.

4 INDEXING TRAJECTORY DATA

We outline three trajectory indexing techniques for the GPU. Although our implementations use OpenCL, hereafter we use the more common CUDA terminology to describe our algorithms (GPU as opposed to device, kernel as opposed to program, thread as opposed to work-item, etc.).

4.1 Spatial Indexing: Flatly Structured Grids

Previous work has proposed "flatly structured grids" (FSG) to index trajectory data spatially on the GPU [21]. In that work, the authors focus on 2-D spatial data (and Hausdorff distance) while our context is 3-D spatiotemporal data (and Euclidean distance). An interesting question is whether spatial indexing with FSGs is effective even when the data has a temporal dimension. In what follows we describe an FSG indexing scheme and accompanying search algorithm for the GPU. We call this approach GPUSPATIAL.

4.1.1 Trajectory Indexing

A FSG is a 3-D rectangular box with $grid_x$, $grid_y$, $grid_z$ cells in the x, y, and z spatial dimensions, respectively, for a total of $grid_x \times grid_y \times grid_z$ cells. Each line segment l_i in D is contained in a spatial MBB defined by two points MBB_i^{min} and MBB_i^{max}, where $MBB_i^{min} = (\min(x_i^{start}, x_i^{end}), \min(y_i^{start}, y_i^{end}), \min(z_i^{start}, z_i^{end}))$

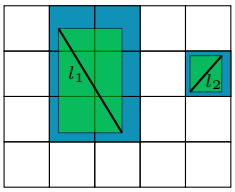

Fig. 1: 2-D example rasterization of two line segment MBBs (green) to grid cells (blue) in a 4×5 FSG. l_1: a long line segment whose MBB spans six grid cells; l_2: a short line segments whose MBB spans one grid cell.

and $MBB_i^{max} = (\max(x_i^{start}, x_i^{end}), \max(y_i^{start}, y_i^{end}), \max(z_i^{start}, z_i^{end}))$, and assigned to the FSG by rasterizing its MBB to grid cells. Figure 1 shows a 2-D example for two line segments and a 5×4 FSG. Each line segment may occupy more than one grid cell, and some grid cells can remain empty. 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 linearized coordinate computed from the cell's x, y, and z coordinates using row-major order.

Each cell C_h is defined by h, and by an index range $[A_h^{min}, A_h^{max}]$ in an additional integer "lookup" array, A. $A[A_h^{min} : A_h^{max}]$ contains the indices of the line segments whose MBBs overlap cell C_h (the notation $X[a : b]$ is used to denote the "slice" of array X from index a to index b, inclusive). In other terms, if l_i's MBB overlaps C_h, then $i \in A[A_h^{min} : A_h^{max}]$. Since the MBB of line segment l_i can overlap multiple grid cells, i can occur multiple times in array A. Section 1 in the supplemental material discusses an example of this indexing scheme, showing the relationship between arrays D, G, and A.

One of the objectives of the above design is to reduce the memory footprint of the index. This is why we only index non-empty grid cells, and why for each cell C_h we do not store its spatial coordinates but instead compute h whenever needed (thereby trading off space for time). Furthermore, the use of lookup array A makes it possible for array G to consist of same-size elements (even though some cells contain more line segments than others). Without this extra indirection through array A, it would have been necessary to store entry segment ids directly into the elements of G. This, in turn, would have made it necessary to pick an element size large enough to accommodate the cell with the largest number of entry segments, thereby wasting memory. D, A, and G are stored in GPU memory before query processing begins.

4.1.2 Search Algorithm

The trajectory segments in Q are not sorted by any spatial or temporal dimension. This is because sorting segments temporally would not be effective when using a spatial index. Regarding spatial sorting, it is not clear

by which dimension the segments should be sorted. 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 to achieve meaningful contiguity for arbitrary 3-D spatial data. As a result, we simply store segments that are part of the same query trajectory contiguously. Each query segment q_k is assigned to a GPU thread. The kernel first calculates the MBB for q_k and the FSG cells that overlap this MBB. Given the x, y, z coordinates of each such cell in the FSG, the kernel computes its linearized coordinate (h) using a row-major order. A binary search is used to find whether cell C_h occurs in array G, in $O(\log |Q|)$ time. In this manner the kernel creates a list of non-empty cells that overlap q_k's MBB. For each cell C_h in this list, the indices of the entry segments it contains are computed as $A[A_h^{min} : A_h^{max}]$. These indices are appended to a buffer U_k.

The rationale for the above scheme is that with a spatial indexing scheme there is no good approach for storing index entry segments in a contiguous manner (since one would have to arbitrarily pick one of the spatial dimensions). This is why we must resort to using buffer U_k as opposed to, for instance, a 2-integer index range in a contiguous array of entry segments. Each entry in U_k is then compared to the query segment q_k to see if it is within the threshold distance. Note that while the segments are expected to be relatively nearby each other spatially (given their FSG overlap), they may not overlap temporally. Note that we do not remove duplicate indices in buffer U_k, leading to some redundant entry segment processing. Removing duplicates would amount to sorting buffer U_k, as done for instance in [21], which thus comes at an additional computational cost that, as shown in our experimental results, offsets the benefits of removing redundant segment processing.

The use of buffer U_k creates memory pressure and its size must be defined prior to the search (see Section 3). We define an overall buffer size, s, that is split equally among all queries ($|U_k| = s/|Q|$). If the processing of query q_k exceeds the capacity of U_k, then the thread terminates, and stores the query id into an array that is sent back to the host. Once the kernel execution finishes, the host re-attempts the execution of those queries that could not complete due to memory pressure. For each such re-attempt, memory pressure is lower because fewer queries are executed (i.e., $|U_k|$ is larger). It is likely that more efficient methods could be designed, but our results show that completely discounting the overhead of these re-attempts does not significantly change how GPUSPATIAL compares to the schemes we propose in upcoming sections.

The pseudo-code of the search algorithm is shown in Algorithm 1. Its arguments are: (i) the FSG array (G);

Algorithm 1 GPUSPATIAL kernel.

```
1:  procedure SEARCHSPATIAL(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
```

(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 between the host and the GPU, either as input or output, are shown in boldface. Other arguments are either pointers to pre-allocated zones of (global) GPU memory or integers. The algorithm first checks 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 6-8). 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 to the query (line 14) and atomically adds any query result to the result set (line 16). Once all 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.

4.2 Temporal Indexing

In this section, we propose a purely temporal partitioning strategy, GPUTEMPORAL. The indexing scheme is similar to that used in [29], and we described it here for completeness. The search algorithm, however, is very

different due to different memory constraint assumptions.

4.2.1 Trajectory Indexing

We begin by sorting the entries in D by non-decreasing t_{start} values, re-numbering the entry segments in this order, i.e., $t_i^{start} \leq t_{i+1}^{start}$. The full 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 divide this full temporal extent so as to create m logical bins of fixed length $b = (t_{max} - t_{min})/m$. We assign each entry segment, l_i, $i = 1, \ldots, |D|$, to a bin, where l_i belongs to bin B_j, $j = 1, \ldots, m$, if $\lfloor t_i^{start}/b \rfloor = j$. There can be temporal overlap between the line 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 $B_j^{end} = \max((j+1) \times b, \max_{l_i \in B_j} t_i^{end})$. B_j^{start} does not depend on the line segments in bin B_j, but B_j^{end} does. The temporal extent of bin B_j is defined as $[B_j^{start}, B_j^{end}]$. Given the definitions of B_j^{start} and B_j^{end}, the union of the temporal extents of the bins is equal to the full temporal extent of D. 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. Bin B_j is thus fully described as (B_j^{start},B_j^{end}, B_j^{first}, B_j^{last}). The set of bins forms the temporal database index. Section 2 in the supplemental material discusses an example of this indexing scheme, showing how line segments are assigned to temporal bins.

4.2.2 Search Algorithm

Before performing the search, the following pre-processing steps must be performed. First, 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 calculate the index range of the contiguous bins that it overlaps temporally. A naïve algorithm for computing this overlap would be to scan all bins in $O(m)$ time. A binary search could be used to obtain a logarithmic time complexity. In practice, however, there are many temporally contiguous query segments and each overlaps only a few bins. Since segments in Q are sorted by non-decreasing t_{start} values, the search can be done efficiently by using the first temporal bin that overlaps the previous query segment as the starting point for the scan for the temporal bins that overlap the next query segment. The search thus typically takes near-constant time. Let $\mathcal{B}_k$ denote the set of contiguous bins that temporally overlap query segment q_k, as identified by the above search. In constant time we compute the index range of the entry line segments that may overlap and must be compared with 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 the *schedule*, S. Each GPU thread compares a single query to the line segments in D whose indices are in the E_k range.

In all of our experiments, the time to compute S on the CPU is a negligible portion of the overall response time. Over all experiments for the largest dataset used in this work (*Merger*, as described in Section 5), the schedule computation on the CPU accounts for at most 0.002% of the compute time on the GPU.

Algorithm 2 GPUTEMPORAL kernel.

```
1:  procedure SEARCHTEMPORAL(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. Its arguments are: (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 host-GPU transfers 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.

4.3 Spatiotemporal Indexing

In the two previous sections, we have proposed a purely spatial and a purely temporal indexing scheme. The spatial scheme leads to segments in Q and D being compared that are spatially relevant but may be temporal misses (no temporal overlap). Likewise, the temporal indexing scheme compares temporally relevant segments in Q and D, but these segments may be spatial misses (no spatial overlap). Therefore, either approach can outperform the other depending on the spatiotemporal characteristics of Q and D. Assuming for the sake of discussion that these characteristics do not give any such particular advantage to either one of the two indexing approaches, we can reason about their relative performance. First, the spatial indexing approach requires buffer space to store the spatially overlapping trajectory segments. In contrast, because the temporal indexing scheme is indexed in a single dimension, the temporally overlapping entry segments can be defined by an index range

in D, which represents significant memory space savings. The same method could possibly be used with a spatial indexing scheme if considering only one of the spatial dimensions, making the index no longer a multi-dimensional grid, but instead a linear array. This approach would however drastically decrease the spatial selectivity of the search, leading to large increases in wasted computational effort (i.e., comparisons of segments that have no overlap in one or two of the spatial dimensions). Second, to minimize the memory footprint on the GPU, the spatial scheme requires two additional arrays (G and A), thus leading to two indirections in global GPU memory. In contrast, the temporal scheme requires a single indirection. Moreover, the entry segments are stored contiguously in the temporal scheme, while this is not the case in the spatial scheme.

We propose an alternate spatiotemporal indexing scheme, GPUSPATIOTEMPORAL, that retains the benefit of both GPUSPATIAL and GPUTEMPORAL, without some of the above drawbacks.

4.3.1 Trajectory Indexing

GPUSPATIOTEMPORAL adopts a temporal index so as to avoid the buffering and multiple indirection issues of spatial indexing, but subdivides each temporal bin into spatial subbins to achieve spatial selectivity. Entry segments in D are assigned to m temporal bins exactly as for GPUTEMPORAL. We then compute the spatial extent of D in each dimension. For instance, in the x dimension the extent of D is:

$$[x_{min}, x_{max}] = [\min_{l_i \in D}(\min(x^i_{start}, x^i_{end})) \ , \quad \max_{l_i \in D}(\max(x^i_{start}, x^i_{end}))] \ .$$

Spatial extents in the y and z dimensions are computed similarly. We then compute the maximum spatial extent in each dimension of the entry segments, which for the x dimension is $\max_{l_i \in D} |x^i_{start} - x^i_{end}|$. Maximum spatial extents are computed similarly for the y and z dimension. For each of the temporal bins, we create v spatial subbins along each dimension, 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 expressed as $v \leq (x_{max} - x_{min}) / \max_{l_i \in D} |x^i_{start} - x^i_{end}|$. We place this constraint for two reasons: (i) to eliminate duplicates in the result set, and (ii) to reduce the amount of redundant information in the index. In total we have $m \times v$ subbins and we denote each subbin as $\hat{B}_{i,j}$, with $i = 1, \ldots, m$ and $j = 1, \ldots, v$.

The above indexing of line segments to temporal and spatial bins is implemented via three integer arrays, X, Y, and Z. Each array stores the ids of the line 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 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. The reason for storing the ids in this manner is as follows. Consider a query segment with some spatial and temporal extent. This query may overlap several contiguous temporal bins (as shown in Section 4.2). However, because of the way in which we choose the sizes of the spatial subbins, 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 $0 \leq i \leq v$. 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. When compared to the purely temporal index, this spatiotemporal indexing scheme requires only additional space in GPU memory for the X, Y, and Z integer arrays, which corresponds to $\gtrsim 3|D| \times 4$ bytes. Section 3 in the supplemental material discusses an example of this indexing scheme, showing how segments are assigned to temporal and spatial bins and how arrays X, Y, and Z are constructed.

4.3.2 Search Algorithm

On the host, as for GPUTEMPORAL, we first sort Q and for each query segment calculate the temporally overlapping entries from the temporal bins. We also compute the set of spatially overlapping subbins in each dimension. This computation also takes place on the host, where the description of the bins and subbins are stored. Arrays X, Y, and Z are stored on the GPU. The obvious option would be to compute the intersection of entry segments that belong to these subbins so as to select only spatially relevant entry segments and sent a list of their indices to the GPU. Unfortunately, this is an unpractical approach due to the memory footprint of the list for large and or dense datasets and for relevant query distances. Let us consider the *Random-dense* and *Merger* datasets used in this work (see Section 5). For *Random-dense* the list occupies 52MiB for query distance $d = 0.1$ and 9,353MiB for $d = 0.9$. For *Merger* the list occupies 293MiB for $d = 1$ and 3,018MiB for $d = 9$. These sizes assume that the database index has perfect selectivity so that the list only contains indices of line segments that will be part of the result set. However, a state-of-the-art spatiotemporal R-tree produces a candidate set that is 10 times larger than the result set for 3-d trajectories (see Figure 8 in [32]). Therefore, it is reasonable to expect that the list would be roughly one order of magnitude larger than the aforementioned sizes. In practice, besides the overhead of sending large amounts of data from the host to the GPU, the memory footprint of the

index list is thus prohibitive due to the limited global memory capacity of the GPU (in our case 5GiB).

Since it is not feasible to use a list of indices, instead we use an approach that uses a fixed and small number of indices. Among the three spatial dimensions we pick the one in which the number of entry segments that overlap the query segment is the smallest. We then simply send to the GPU an index range, 2 integers, in the X, Y, or Z array, depending on the dimension that was picked. This approach has low overhead and memory footprint. Its drawback is that it can lead to wasteful computation on the GPU (i.e., evaluation of entry segments that do not overlap with the query segment in one of the other two spatial dimensions). Our results show that the search algorithm achieves good performance in spite of these wasteful computation, as might be expected given the GPU's sheer computational power.

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 (to preserve alignment). GPUSPATIOTEMPORAL requires 1 extra indirection in comparison to GPUTEMPORAL, and avoids storing the overlapping entry indices in a buffer like in GPUSPATIAL. We then sort S based on the lookup array specification so as to minimize thread serialization due to branching. Over all experiments for the largest dataset used in this work (*Merger*, as described in Section 5), the schedule computation on the CPU accounts for at most 0.04% of the compute time on the GPU.

As explained in the previous section, we enforce a minimum size for the spatial subbins. Ensuring that subbins are not too small is necessary for two reasons. First, with small subbins each entry segment could overlap many subbins with high probability. As a result, the query id would occur many times in arrays X,Y, and/or Z, thereby wasting memory space on the GPU and causing redundant calculations. Second, given our indexing scheme and search algorithm described hereafter, a query that overlaps multiple subbins along all three spatial dimensions may lead to duplicates in the result set. These duplicates would then need to be filtered out (either on the GPU or the CPU). To avoid duplicates, we simply default to the purely temporal scheme whenever duplicates would occur. While this behavior wastes computation (i.e., we lose spatial filtering capabilities), the constraint on subbin size described in the previous section ensures that it occurs with low probability.

The pseudo-code of the search algorithm is shown in in Algorithm 3. Its arguments are: (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 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 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, it 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 (line 10) as in Algorithm 2.

Algorithm 3 GPUSPATIOTEMPORAL kernel.

```
1:  procedure SEARCHSPATIOTEMPORAL(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
```

5 EXPERIMENTAL EVALUATION

5.1 Datasets

We evaluate the performance of our various indexing methods for 3 4-dimensional datasets:

Random-1M – A small, sparse synthetic dataset 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. This dataset is representative of small and sparse datasets in which few or no entry segments are expected to lie within distance d of a query segment, i.e., with a low number of *interactions*.

Merger – A large, real-world dataset[1] from the field of astrophysics, which 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.

1. This dataset was obtained from Josh Barnes [33].

Random-dense – A synthetic dataset motivated by astrophysics applications but denser than *Merger*, 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. [34], $n_\odot = 0.112$ stars/pc^3. *Random-dense* has the same number of particles as one disk in the *Merger* dataset (65,536) and 193 timesteps, yielding 12,582,912 entry segments, but matching the density of [34]. This requires a cubic volume of $65536/0.112 = 585142$ pc^3, i.e., a cube with length, width and height of 83.64 pc. We generate trajectories as random walks as in the *Random-1M* dataset, where all of the particles are initially populated within the aforementioned cube. We allow the trajectories to move a variable distance in each of the 3 spatial dimensions at each timestep (between 0.001 and 0.005 kpc). If a particle moves outside of the cube by 20% of the length of the cube in any dimension, the particle is forced back towards the cube. The particles, on average, cannot travel too far from the cube such that we maintain a roughly consistent trajectory density at each timestep. This dataset aims to represent a density consistent within the range of possible densities within the Milky Way. Note that increasing the trajectory density by several factors (e.g., 4-fold) would still be consistent with that resembling the disk in the inner Galaxy. So although we call this dataset "dense," even denser datasets are relevant in the application domain.

5.2 Experimental Methodology

For all our distance threshold search implementations the GPU-side is developed in OpenCL and the host-side is developed in C++. The GPU-side implementation runs on an Nvidia K20c card (Kepler microarchitecture) with 5GiB of RAM and 2496 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 query response time as an average over 3 trials (standard deviation over the trials is negligible). For each experiment, given the memory capacity of our card, the memory footprint of the dataset, and the memory footprint of the index, we allocate on the GPU a buffer for holding the result set of the search that is as large as possible. The size of this buffer is given for all the experiments described in the upcoming sections. When this buffer is overcome, then the query set is processed incrementally via multiple kernel invocations, as explained in Section 3. The reported response times include the induced overhead of these invocations and corresponding data transfers. The response time does not include the time to build the index or the time to store D and the index in GPU memory. These operations can be performed off-line before query processing begins.

We consider three experimental scenarios, each for one of our datasets: (S1) The *Random-1M* 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; and (S3) The *Random-dense* dataset and a query set with 265 trajectories each with 193 timesteps for a total of 50,880 query segments. For each scenario, we use ranges of query distances (in units of kpc for S2 and S3).

In addition to our GPU implementations we also evaluate a CPU-only implementation, CPU-RTREE. This implementation relies on an in-memory R-tree index [11], and is multithreaded using OpenMP. Threads traverse the R-tree in parallel, each for a different query segment, and return candidate entry segments. This implementation was developed in our previous work [16], [32]. All executions of CPU-RTREE use 6 threads on our 6-core CPU and achieve high parallel efficiency [29]. Like for the GPU implementation, our response time measurements do not include the time to build the index tree. One important driver of response time for index trees is how trajectory segments are assigned to MBBs [16], [31], [35]. CPU-RTREE stores $r \geq 1$ segments per MBB. There is a trade-off between the time to search the index (which decreases as r increases due to lower tree depth) and the time to process the candidate (which increases as r increases due to higher index overlap).

Although the experimental results in the following sections are constrained by the specifics of our platform, the results for CPU-RTREE are used to demonstrate that the GPU can be used efficiently for distance threshold searches. Note that a fundamental difference between CPU-RTREE and our GPU implementations is that the former relies on index-tree traversals while the latter relies on non-hierarchical indexes.

5.3 Results for the *Random-1M* Dataset

Figure 2 shows response time vs. the number of entry segments per MBB (r) for CPU-RTREE for a range of d values. These results illustrate the trade-off mentioned in the previous section: neither using $r = 1$ or using a large value of r leads to the lower response time. Several values in between lead to good response time across all query distances, e.g., $r = 10$.

Figure 3 plots response time vs. d for GPUSPATIAL for a range of grid resolutions (i.e., numbers of grid cells). In all GPU implementations on this dataset, we allocate a buffer for the result set having 5×10^7 elements. We use a total buffer size, $|U|$, of 2GiB to store overlapping entry segments, which is larger than the space necessary to store D. This is thus an optimistic configuration for the FSG index. The results show that using too few grid

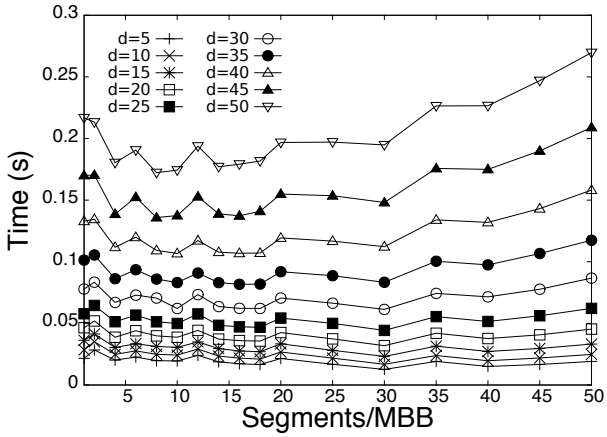

Fig. 2: Response time vs. number of entry segments per MBB (r) for CPU-RTREE in scenario S1 with $d = 5, 10, \ldots, 50$.

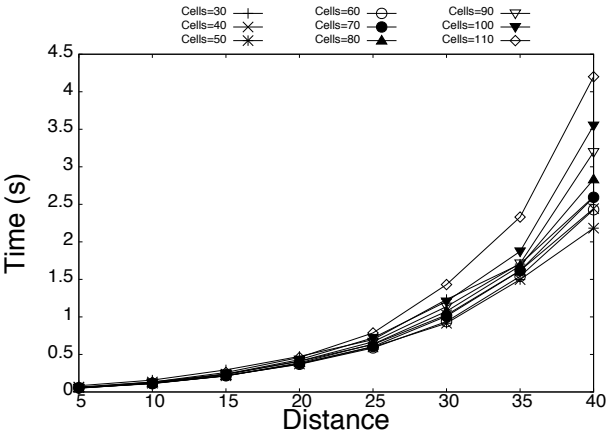

Fig. 3: Response time vs. d for GPUSPATIAL in scenario S1. Different curves are shown for different numbers of spatial cells in the x, y, and z dimensions (i.e., "Cells=10" means a $10 \times 10 \times 10$ grid).

cells leads to poor performance due to poor spatial selectivity meaning that: (i) a large candidate set must be processed and (ii) many GPU threads overflow their entry buffers (U_k) thus requiring multiple query processing attempts. Conversely, using too many grid cells also leads to poor performance because entry segments overlap multiple cells, causing duplicate index entries, and thus duplicates in the result set. Although filtering out these duplicates takes negligible time, transferring them from the GPU back to the host incurs significant overhead. In these experiments, for FSG resolutions between 30 and 110 in increments of 10, using 50 cells per dimension leads to the lowest response time. Regardless of the FSG resolution, we see rapid growth in response time as d increases, a behavior already mentioned in [21]. While FSG indexes have been documented to perform well for purely spatial data and/or for point searches (rather than line segment searches), we find that for

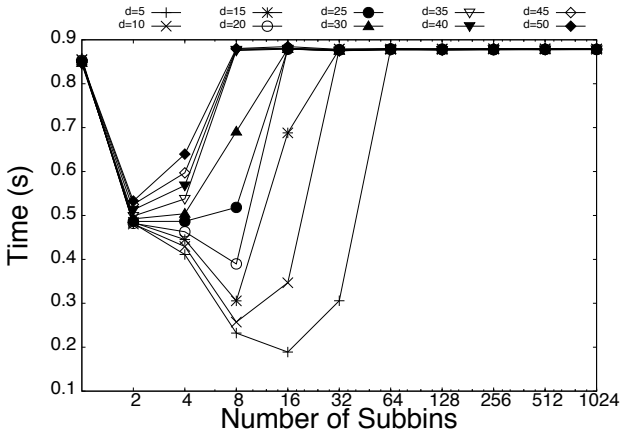

Fig. 4: Response time vs. the number of subbins (v) for GPUSPATIOTEMPORAL in scenario S1. The number of temporal bins is 10,000. Different curves are shown for different query distances ($d = 5, 10, ..., 50$).

spatiotemporal trajectory searches they are very sensitive to the query distance.

The behavior of GPUTEMPORAL is defined by the number of temporal subbins used to construct the index (on the host). Our results, not shown, clearly show the expected trade-off. Using too few temporal bins leads to insufficient temporal discrimination, resulting in wasteful interactions that negatively impact the response time. But as the number of bins increases the response time converges to a minimum value. For this dataset, we find that using more than 5,000 bins does not lead to further response time reductions. But a conservative approach that would pick, e.g., 10,000 bins, does not experience any response time increase.

Figure 4 shows response time vs. the number of subbins for GPUSPATIOTEMPORAL, using 10,000 temporal bins, for several d values. For low d a greater number of spatial subbins is desirable. This is because it is unlikely that a query will overlap multiple subbins, which would cause our algorithm to revert to the purely temporal method that has no spatial selectivity. As d increases, queries overlap multiple spatial subbins with higher probability. As a result, better performance is achieved with fewer subbins. Recall that we require that a query fall within a single subbin so as to avoid duplication in the result set. Without this requirement, an increasing number of subbins would suggest an increase in the duplication of entries in the index, thereby increasing the number of candidates that need to be processed (the same trade-off discussed for GPUSPATIAL). There is thus a trade-off between having too few or too many subbins, even when duplicates in the result set are permitted.

Figure 5 shows response time vs. d for our four implementations. Each implementation is configured with good parameter values based on previous results in this section (see the caption of the figure). The

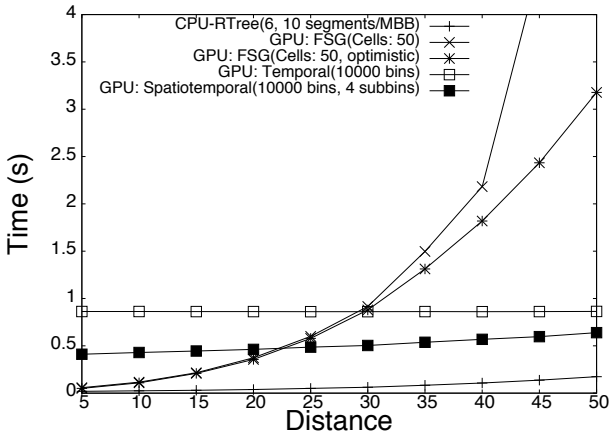

Fig. 5: Response time vs. d for scenario S1. For CPU-RTREE we use $r = 10$ segments/MBB; for GPUSPATIAL we use 50 cells per spatial dimension; for GPUTEMPORAL we use 10,000 bins; and for GPUSPATIOTEMPORAL we use 10,000 temporal bins and $v = 4$ spatial subbins: For GPUSPATIAL we plot an optimistic curve that ignores kernel re-launch overheads.

first observation is that CPU-RTREE is best across all query distances. Comparing the GPU implementations, we see that GPUSPATIAL performs better than GPUTEMPORAL and GPUSPATIOTEMPORAL when $d < 20$, but that it does not scale well for larger d values. One may wonder whether this lack of scalability comes from the overhead of re-launching the kernel due to buffer overflows. Figure 5 plots an "optimistic" curve that discounts this overhead. We see that the same trend, albeit not as extreme, remains. Consequently, GPUSPATIAL's poor scalability is intrinsic and not solely due to the need to relaunch kernels due to memory constraints. The temporal and spatiotemporal indexing methods have consistent response times across query distances. Because this dataset is sparse, even with large query distances the size of the result set is small. Consequently, the fraction of the execution time of GPUTEMPORAL and GPUSPATIOTEMPORAL spent identifying and adding items to the result set is low across all query distances in these experiments. This overhead becomes noticeable for denser datasets, as seen in upcoming sections. For GPUSPATIOTEMPORAL we could have selected the best number of subbins for each value of d from Figure 4, which would have improved the results (instead we have used $v = 4$ spatial subbins per temporal bin). Nevertheless, we see that GPUSPATIOTEMPORAL outperforms GPUTEMPORAL, showing that the use of a more complex indexing scheme (i.e., with one more indirection to implement spatial selectivity) yields performance benefits.

One possible reason for the poor performance of our GPU algorithms relative to the CPU algorithm could be that their executions are global-memory-bound. Note that a common optimization on the GPU is to ensure

that global memory accesses are coalesced [36]. Such coalescing is challenging in our case, due to the fact that our computation is data-dependent. In general, given a query, it is not possible to determine which entries it will overlap, and thus it is not possible to enforce that these entries be stored contiguously in memory. As a result, our implementations lead to many uncoalesced accesses. We have measured the effective global-memory throughput of our algorithms by dividing the total number of bytes loaded from and stored to global memory on the GPU by the kernel execution time. The throughput of GPUTEMPORAL does not depend on d, while the throughput of GPUSPATIOTEMPORAL increases slightly as d increases. Regardless, across all our experiments their global memory throughput is below 15 GiB/s. This is less than 10% of the available global memory bandwidth, which is above 200 GiB/s on our GPU. The relatively poor performance of our GPU algorithms is thus not due to a global memory bottleneck on the GPU (even with uncoalesced accesses).

We conclude that one should use an in-memory R-tree on the CPU for small and sparse datasets since the overhead of using the GPU is too large given that few interactions need to be computed.

5.4 Results for the *Merger* Dataset

In this section, we present results for our largest dataset, *Merger* (over 25 million entry segments). The results from Section 5.3 show that GPUSPATIAL does not fare well for large query distances because it does not use any temporal selectivity and thus computes too many interactions. With the *Merger* dataset, this lack of temporal selectivity is harmful even for small query distances. Consequently, for *Merger*, GPUSPATIAL leads to response time much higher than that of the other implementations and we thus omit its results in all that follows.

When running CPU-RTREE on this dataset, regardless of the query distance, we find that storing more than $r = 1$ segments per MBB leads to higher response time, which is unlike what is observed for the *Random-1M* dataset. A higher r value decreases the time to search the R-tree index, but this benefit is offset by the increase in candidate set size. With the large dataset, there are simply too many candidates to justify increasing the overlap in the index. This is an important result. There is a literature devoted to assigning trajectory segments to MBBs for improving response time [16], [31], [35]. These works, however, do not consider large datasets. Our results with a large dataset indicates that deciding how to group multiple trajectory segments into MBBs is not a worthwhile pursuit. On the contrary, our results may even suggest an opposite approach that would splice individual polylines to increase the size of the dataset (which can be thought of as setting $r < 1$).

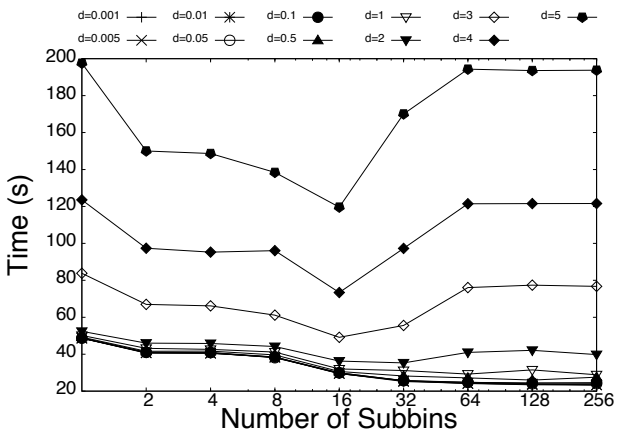

Fig. 6: Response time vs. the number of subbins (v) for GPUSPATIOTEMPORAL in scenario S2. The number of temporal bins is 1,000. Different curves are shown for different query distances between $d = 0.001$ and $d = 5$.

We do not show results for GPUTEMPORAL as they are similar to those for the *Random-1M* dataset. For this dataset, using 1,000 temporal bins leads to the lowest response time, which is consistent across all query distances. For GPUTEMPORAL, a result set buffer was allocated to store 5×10^7 elements. Figure 6 shows response time vs. number of subbins for GPUSPATIOTEMPORAL, where 1,000 temporal bins are used. For GPUSPATIOTEMPORAL, the allocated size of the buffer for the result set is 4×10^7 elements, which is lower than the size of the buffer for GPUTEMPORAL, as there is additional space required to store the index. Curves are shown for several d values. We observe that using $v = 16$ subbins leads to good results across all query distances, and this value is in fact best for most query distances we have attempted. While Figure 4 shows a dependency between v and d for the *Random-1M* dataset, this dependency does not exist for a large dataset with many interactions. The implication is that picking a good v value is likely straightforward for such datasets.

Figure 7 compares the performance of CPU-RTREE and GPUTEMPORAL and GPUSPATIOTEMPORAL (recall that GPUSPATIAL is omitted due to high response time). Each method is configured with good parameter values based on previous results in this section as described in the caption. GPUSPATIOTEMPORAL outperforms GPUTEMPORAL across the board, with response times at least 17.6% faster. At low query distances CPU-RTREE yields the lowest response times. It is overtaken by GPUSPATIOTEMPORAL at $d \sim 1$. At $d = 0.001$ the response time for the CPU implementation is 9.70 s vs. 41.75 s for GPUSPATIOTEMPORAL (the GPU implementation is 203.8% slower). At $d = 5$ these response times become 184.4 s, and 119.61 s, respectively (the GPU implementation is 54.2% faster). Note that the increase in response time for GPUTEMPORAL and

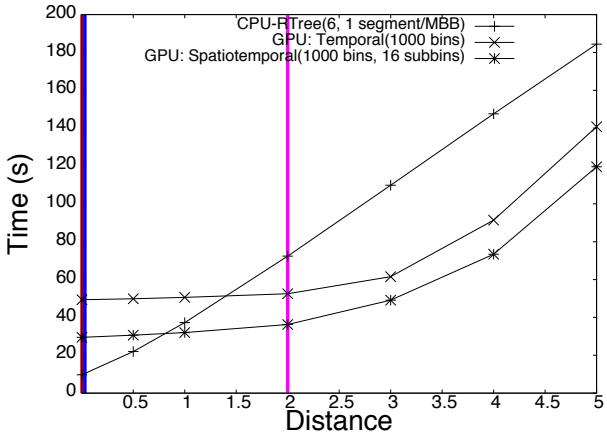

Fig. 7: Response time vs. d for our implementations for scenario S2. For CPU-RTREE we use $r = 1$ segments/MBB; for GPUTEMPORAL, we use 1,000 bins; for GPUSPATIOTEMPORAL, we use 1,000 temporal bins and $v = 16$ spatial subbins. We indicate three distance thresholds relevant for the study of the habitability of the Milky Way. Red: close encounters between stars and planetary systems [37]; Blue: supernova events on habitable planetary systems [3], and Magenta: studying the effects of gamma ray bursts on habitable planets [38]. Both the Red and Blue lines are close to the vertical axis.

GPUSPATIOTEMPORAL for query distances $d > 4$ is due to the overhead of communication between the host and the GPU. For these large query distances, the result set is too large to fit in the buffer allocated on the GPU, thus requiring multiple kernel invocations. The expectation is for this overhead to be reduced in the future as bandwidth between the host and the GPU improves and as GPU memory becomes larger.

As in the previous section, we compute effective global memory throughputs. For both algorithms the throughput decreases as d increases, and across all experiments the throughput of GPUTEMPORAL, resp. GPUSPATIOTEMPORAL, is below 16 GiB/s, resp. 10 GiB/s. As for the *Random-1M* dataset, only a small fraction of the available global memory bandwidth is used, due to the compute-bound moving distance calculations.

Overall, we conclude from these results that the GPU implementation outperforms CPU-RTREE when using large datasets or when sufficiently large query distances are considered.

5.5 Results for the *Random-dense* Dataset

We now present results for the *Random-dense* dataset. Recall that for the small *Random-1M* dataset, using $r = 10$ segments per MBB leads to good response time across all query distances, while for the larger *Merger* dataset using $r = 1$ was best due to the large number of interactions. The size of *Random-dense* is in between that of *Random-1M* and *Merger*, and we find empirically that using $r = 4$ leads to low response time across all query distances. As in the previous section, we do not show results for GPUSPATIAL due to very high response time.

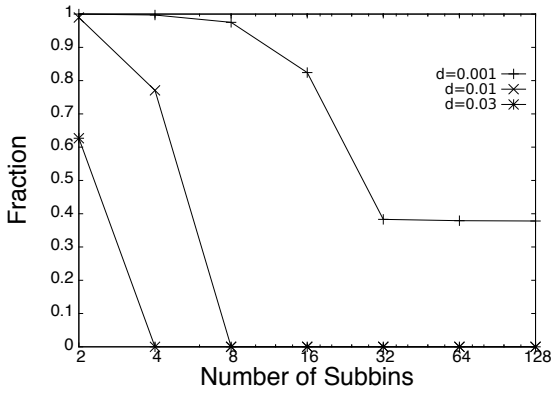

Fig. 8: Fraction of queries that use the entries provided by subbins vs. the number of subbins (v).

We do not show results for GPUTEMPORAL because they are similar to those for the two previous datasets: using 1,000 temporal bins leads to the lowest response time, which is consistent across all query distances.

As for the *Merger* dataset, the response time of GPUSPATIOTEMPORAL for the *Random-dense* dataset is not sensitive to the number of subbins in use. However, we found that with this dataset the use of subbins for reducing response time is only possible for small query distances ($d = 0.001, 0.01, 0.03$). This is because the dataset is smaller than *Merger* and because with larger values of d, the queries are more likely to fall within multiple subbins (in which case the algorithm degenerates to a purely temporal scheme). Figure 8 shows the fraction of queries that utilized the entries provided by the subbins for $d = 0.001, 0.01, 0.03$, i.e., the fraction of queries for which GPUSPATIOTEMPORAL does not degenerate to GPUTEMPORAL. Only the smallest query distance, $d = 0.001$, permits usage of the spatiotemporal index across a sizable fraction of the number of subbins. For instance for $d = 0.03$ and $v = 2$, just over 60% of the queries use the spatiotemporal index over the pure temporal index, and when $v = 4$, the entries provided by the spatiotemporal index are not used.

Given the density of the dataset, 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. For GPUTEMPORAL, and GPUSPATIOTEMPORAL, the size of the buffers are increased to 9.8×10^7, and 9.2×10^7 items, respectively.

This increase in buffer size for *Random-dense* lowers the number of kernel invocations, and thus leads to decreases in response time. For instance, at $d = 0.09$ (which requires the greatest number of kernel invocations), the spatiotemporal index, with $v = 2$, using the larger buffer for the result set leads to a response time that is

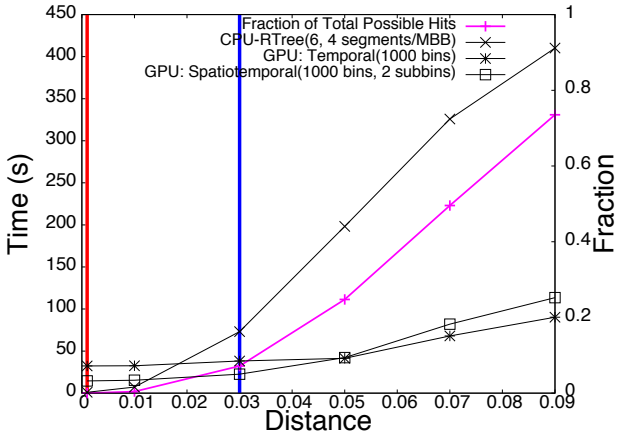

Fig. 9: Response time (left vertical axis) and fraction of entries within distance d of the query (right vertical axis) vs. d for CPU-RTREE, GPUTEMPORAL, and GPUSPATIOTEMPORAL for scenario S3. For the CPU we show results for $r = 4$. 1,000 temporal bins are used for both the temporal and spatiotemporal indexing methods. $v = 2$ spatial subbins are used for the spatiotemporal indexing method.

59.63% lower than that with the initial buffer. Although we could not run experiments with a larger buffer size for scenario S2 (due to the size of the *Merger* dataset), we would expect similar performance gains.

Figure 9 shows response time vs. d for CPU-RTREE and with the larger buffer sizes for GPUTEMPORAL and GPUSPATIOTEMPORAL. The query distance range spans a wide range of result set sizes. We span a range of scenarios with respect to the number of entries that are within the query distance, as shown on the right vertical axis. When $d = 0.001$, $\approx 0\%$ of the entries are within the query distance, and when $d = 0.09$, 73.9% of the entries are within the query distance. CPU-RTREE outperforms the GPU implementations only for very small query distances $d \lesssim 0.02$. For $d > 0.05$, GPUSPATIOTEMPORAL performs slightly worse than GPUTEMPORAL. This suggests that for dense datasets and large query distances, a purely temporal indexing scheme performs best. At $d = 0.05$, GPUTEMPORAL is 380% faster than CPU-RTREE (with $r = 4$).

The effective global memory throughput of our algorithms decreases as d increases and is low (below 15 GiB/s for both algorithms), showing once again that our implementations are compute-bound.

Comparing Figures 7 (*Merger*) and 9 (*Random-dense*) shows that the range of query distances for which the GPU method is preferable to the CPU method is much larger for the *Random-dense* dataset (consider the query distances for relevant application scenarios – the red, blue, and magenta vertical lines). In the astrophysics domain datasets denser than the *Random-dense* dataset are relevant (i.e., to study the galactic regions at $R < 8$ kpc). For such datasets a GPU approach will provide even more improvement over a CPU implementation.

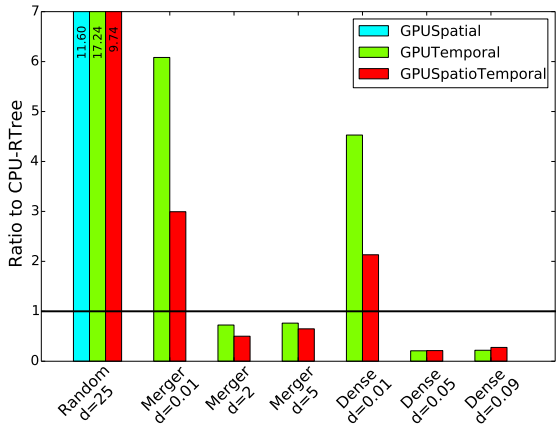

Fig. 10: Ratio of GPU to CPU response times for various datasets and query distances. Values below the $y = 1$ line indicate improvements over CPU-RTREE.

6 CONCLUSIONS

We have proposed indexing methods and search algorithms for distance threshold similarity searches over spatiotemporal trajectory datasets on the GPU. To summarize our results, Figure 10 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 observation is 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 very small). For dense datasets and/or large query distances 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 many applications, including our driving application domain. Overall, a spatiotemporal indexing method that achieves both temporal and spatial selectivity, without resorting to an index tree, is effective on the GPU. Future trends for GPU technology (faster host–GPU bandwidth, increased memory, etc.) should provide increasing advantages over CPU implementations. Finally, our experiments show that for the in-memory R-tree CPU implementation, the well-studied question of how to split a trajectory and store it in multiple MBBs may not be pertinent for large datasets as storing a single segment per MBB is appropriate. In fact one may even attempt to splice segments and increase dataset size so as to trade-off higher index-tree search time for lower index overlap. This result should apply to other similarity searches, such as kNN searches.

An interesting avenue for future research is to explore analytical modeling techniques for predicting, for

a dataset and a set of queries, whether a GPU execution of the search would be worthwhile. Because search performance is data-dependent, the challenge is to identify salient metrics that can drive accurate response time models. Another future direction is to investigate hybrid implementations of the distance threshold search that uses both the CPU and GPU for query processing. Finally, a broader future direction is to apply our indexing techniques to other spatial/spatiotemporal trajectory searches.

ACKNOWLEDGMENTS

The authors are grateful to Josh Barnes for providing the *Merger* dataset. This material is based upon work supported by the National Aeronautics and Space Administration through the NASA 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. Gowanlock and H. Casanova, "Indexing of Spatiotemporal Trajectories for Efficient Distance Threshold Similarity Searches on the GPU," in *Proc. of the 29th IEEE International Parallel & Distributed Processing Symposium*, 2015.

[3] 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.

[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, "Nearest neighbor search on moving object trajectories," in *Proc. of the 9th Intl. Conf. on Advances in Spatial and Temporal Databases*, 2005, pp. 328–345.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] J. Kim, W.-K. Jeong, and B. Nam, "Exploiting massive parallelism for indexingmulti-dimensional datasets on the gpu," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 8, pp. 2258–2271, Aug 2015.

[20] 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.

[21] ——, "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.

[22] 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.

[23] L. Luo, M. D. F. Wong, and L. Leong, "Parallel implementation of R-trees on the GPU," in *Proc. of the 17th Asia and South Pacific Design Automation Conf.*, 2012, pp. 353–358.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] M. Gowanlock, "In-memory distance threshold searches on moving object trajectories," Ph.D. dissertation, University of Hawai'i at Mānoa, 2015.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] N. Fauzia, L. N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proc. of Symposium on Code Generation and Optimization*, 2015, pp. 12–22.

[37] 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.

[38] 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.

Michael Gowanlock received the B.Sc., and M.Sc. degrees from Trent University in Peterborough, Canada, in 2008, and 2010 respectively, the Ph.D. degree from the University of Hawai'i at Mānoa in Honolulu, U.S.A., in 2015. He is currently a Postdoctoral Associate at MIT Haystack Observatory. His research interests are in the areas of parallel computing and astrobiology.

Henri Casanova received the B.S. degree from the École Nationale Supérieure d'Électronique, d'Électrotechnique, d'Informatique et d'Hydraulique de Toulouse, France, in 1993, the M.S. degree from the National Polytechnic Institute of Toulouse, France, in 1994, and the Ph.D. degree from the University of Tennessee Knoxville, U.S.A., in 1998. He is currently a Professor in the Information and Computer Science Dept. at the University of Hawai'i at Manoa. His research interests are in the areas of parallel and distributed computing.