2026 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
September 27 - October 1, 2026. Pittsburgh, PA, USA

RobotFD: GPU-Accelerated Fréchet Distance for Robot Path Analysis

Zhehu Yuan, Rubao Lee, and Xiaodong Zhang

***Abstract*— Fréchet distance (FD) measures the similarity between geometric paths while respecting their traversal order and serves as a principled measure for order-aware path comparison. Such order-aware similarity is critical in many robotic tasks, including task space motion planning and control, imitation learning, Simultaneous Localization and Mapping (SLAM), and visual odometry evaluation. However, computing FD remains computationally expensive even with the best existing GPU-optimized solution, hindering its practical use in robotic systems. This high computational cost is largely due to limited parallelization. To address this limitation, we develop RobotFD, a new parallel system, and provide an open source GPU implementation. RobotFD achieves high performance by (i) significantly accelerating the core FD decider on the GPU and (ii) effectively reducing the number of iterations during the search. Our evaluation shows a speedup of up to 39× over the best existing GPU implementation, making FD computation on extremely long paths (approximately one million vertices) practically tractable.**

I. INTRODUCTION

Robot path similarity measures are widely used to align demonstrations in imitation learning [1], [2], [3], to quantify reference path tracking error in motion planning and control [4], [5], [6], to evaluate and align estimated trajectories in SLAM [7], and to quantitatively compare estimated trajectories against reference trajectories in visual and visual-inertial odometry evaluation [8], [9]. Among various similarity notions, the Fréchet distance (FD) provides a principled measure for comparing continuous paths, as it is defined over continuous curves, enforces order preserving correspondences between paths, and captures the maximum deviation along an execution under an optimal reparameterization [10]. For example, in surgical robotic manipulation, an executed end-effector path is often compared against a planned or demonstrated reference path [6]. In this setting, as shown in Fig. 1, an average tracking error may fail to capture local deviations that are critical to safety. FD is well suited for such comparisons because it measures the maximum separation under an order preserving alignment of two continuous paths, making it sensitive to localized deviations.

These robotic applications often generate long path traces or require similarity evaluation at high rates. However, computing FD remains computationally expensive, as the runtime grows super-linearly with path length and quickly becomes prohibitive [11].

As a result, FD based similarity often becomes the primary computational bottleneck [12] and is therefore rarely used in practice. In large scale pipelines, it is often avoided because its runtime can exceed practical limits, despite its appeal as a geometry-faithful measure of path similarity. In smaller scale pipelines, approximations of FD are typically adopted to satisfy runtime constraints, trading accuracy for speed and potentially leading to suboptimal results [4], [6], [13].

The authors are with the Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA.

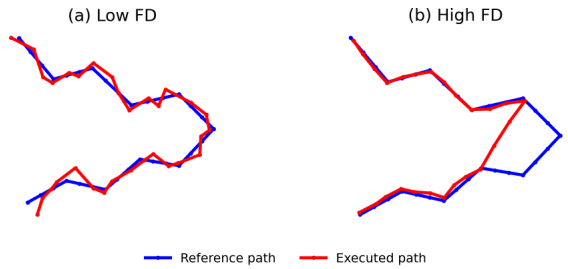


Fig. 1. The blue curve denotes a reference path, and the red curve denotes an executed path. Although the two examples have comparable average tracking errors, the localized deviation in (b) results in a much larger FD than in (a). Such localized deviations can be critical in robotic applications like surgical robotic manipulation.

FD enforces an order preserving alignment that captures how closely the paths follow one another by comparing them as continuous curves rather than discrete samples. Computing this measure typically involves repeatedly invoking a decider to test whether FD is at most a given threshold. This decider demands substantial computational power and introduces strong data dependencies. The existing GPU solution [14] decomposes the computation into two stages: (i) computing the free space intervals (the portions of each line segment that satisfy the distance threshold for each vertex in the other path) independently for all path segment pairs in parallel on the GPU, and (ii) performing reachability propagation sequentially on the CPU, which determines whether a feasible monotone path exists from the start vertices to the end vertices of the two paths. While this design exploits parallelism during the computation of free space intervals, it does not accelerate reachability propagation or the binary search procedure, both of which remain inherently sequential. Moreover, explicitly materializing all free space intervals in GPU memory and transferring them to the CPU introduces additional overhead, and for long paths, this can exceed the available GPU memory capacity, causing the program to fail to complete.

To make FD computation practical for long paths and repeated query workloads that arise in robotics applications, we propose **RobotFD**, a GPU-based FD system for robot path analysis. Although the underlying computation is domain-agnostic, our focus is on robotics workloads, including offline analysis of long paths and high throughput similarity evaluation over candidate robot motions. In **RobotFD**, we

Copyright ©2026 IEEE

leverage the vertex-edge Fréchet distance (VE-FD) [15] as the lower bound to tighten the binary search interval, thereby reducing the number of decider invocations. Furthermore, we implement reachability propagation directly on the GPU in anti-diagonal wavefront order, allowing the alignment frontier to advance in parallel. This design accelerates reachability propagation and eliminates the need to transfer free space intervals to the CPU. To support efficient wavefront propagation, we organize the free space intervals in GPU memory to improve spatial locality and reduce memory access latency. For extremely long path pairs, where storing all free space intervals would exceed device memory capacity, we introduce a memory-bounded computation mode, referred to as Memory-Saving mode. This mode computes free space intervals on demand and discards them after use, thereby eliminating the need to store them and keeping memory usage low, at the cost of additional runtime overhead.

We evaluate **RobotFD** on Cartesian position path traces extracted from four real-world robot trajectory datasets that cover a broad range of robotics settings, from small scale articulated robot motions to large scale ground vehicle and drone trajectories, across both indoor and outdoor environments. We compare **RobotFD** against the best existing GPU implementation and its multicore CPU variant proposed in the same work [14]. **RobotFD** achieves up to 39× end-to-end speedup, averages 15–450 ms per pair for short paths with fewer than 10,000 vertices, and remains practical on extremely long paths with up to approximately one million vertices, for which the baselines fail to complete. We further show that our VE-FD lower bound reduces the number of iterations by 10–30% compared to the best baseline bound, and that our Memory-Saving mode reduces GPU memory consumption to a negligible level at a 13–69% runtime penalty.

Our contributions are listed as follows.

- We propose **RobotFD**, a GPU-based FD system for robot path analysis, together with an implementation[1] that can be easily integrated into robot path analysis pipelines. It supports both latency sensitive querying and offline analysis of extremely long paths.
- We develop a GPU-parallelized VE-FD algorithm and improve its time complexity from $O(mn\log(mn))$ to $O(mn)$, for two paths with m and n vertices, respectively. This algorithm provides a lower bound on FD to tighten the binary search interval, thereby reducing the number of expensive decider calls.
- We accelerate reachability propagation with a GPU-optimized wavefront execution pattern, exploiting parallelism, a locality aware data layout, and a memory saving execution mode to improve throughput.

II. BACKGROUND

A. Path Representation

In robotics, a path is represented as a continuous mapping from a one dimensional progression parameter to a Cartesian position space, describing the trace of a robot's position. Depending on the context, the data may come from a planned motion in configuration or task space, an executed behavior demonstrated by an expert, or a time ordered position estimate produced by a SLAM or odometry system. In practice, such paths are represented as polygonal curves formed by piecewise linear interpolation of sampled states. Throughout this paper, we assume all FD computations are performed on these polygonal representations.

Formally, a path is modeled as a continuous curve:

$$\tau : [0,1] \to \mathbb{R}^d \tag{1}$$

It is represented as a polygonal curve defined by a finite sequence of states connected by linear segments.

B. Fréchet Distance

The Fréchet distance (FD) is a path similarity measure that quantifies how closely two paths resemble each other while respecting their traversal order. Intuitively, FD compares two paths by allowing them to be traversed simultaneously from start to finish, with each path allowed to pause, speed up, or slow down, while keeping continuous forward progression. It measures the minimum achievable maximum separation under an optimal order preserving alignment. As a result, FD captures path similarity in a continuous manner and is sensitive to the ordering of motion, making it well suited for comparing robotic paths where both continuity and order are important.

Formally, given two paths modeled as continuous curves $\tau_1, \tau_2 : [0,1] \to \mathbb{R}^d$, the FD between them is:

$$d_F(\tau_1,\tau_2) = \inf_{\alpha,\beta\in\mathcal{A}} \max_{s\in[0,1]} \|\tau_1(\alpha(s)) - \tau_2(\beta(s))\|_2 \tag{2}$$

where α and β are continuous, non-decreasing reparameterizations of $[0,1]$ with $\alpha(0) = \beta(0) = 0$ and $\alpha(1) = \beta(1) = 1$, and $\mathcal{A}$ denotes the set of all such reparameterizations.

C. The Foundational Fréchet Distance Algorithm

The foundational Fréchet distance algorithm [10] computes FD by iteratively maintaining bounds and invoking a decider. The decider consists of two tightly coupled stages: **(i) free space interval computation** and **(ii) reachability propagation**. After each decider call, the lower or upper bound is updated according to the decision result. If the gap between them is below the target accuracy, their midpoint is returned. Otherwise, the algorithm proceeds to the next iteration.

The **bounding** stage maintains a lower bound L and an upper bound U on FD and selects a sequence of candidate thresholds $\varepsilon \in [L,U]$ to be tested. For each candidate ε, the decider determines whether the two paths can be traversed from start to finish while keeping their separation within ε under a monotonically advancing traversal and order preserving alignment. The decider consists of two stages. In the **free space interval computation** stage, it evaluates, for each cell in the free space diagram induced by the two polygonal paths, the free space of the cell under the distance threshold ε. For two polygonal paths with m and n vertices, the free

[1] https://github.com/ZhehuYuan/RobotFD

space diagram is an $(m-1) \times (n-1)$ grid, where each cell (i, j) corresponds to the pair of segments $\overline{p_i p_{i+1}}$ and $\overline{q_j q_{j+1}}$. The free space intervals of a cell are defined by the parameter range on a segment of one path where its distance to a vertex of the other path is at most ε. In practice, for each cell we store the two endpoints of the free space intervals of the top and right edges of the cell, for a total of four endpoints per cell. In the **reachability propagation** stage, it propagates reachability across adjacent cells to determine whether a feasible monotone path exists from the start cell to the end cell. In existing GPU implementations, interval computation is largely parallel, while reachability propagation is often treated as a bottleneck due to its sequential dependencies and the need to materialize and access a large number of free space intervals.

III. RELATED WORK

A. GPU Acceleration of Fréchet Distance

The prior GPU based effort [14] primarily parallelizes the computation of the free space intervals, since each of them can be computed independently and thus maps naturally onto massively parallel architectures. However, the subsequent reachability propagation over the free space intervals is often implemented as a serial CPU procedure, which limits end-to-end acceleration and incurs additional overhead due to GPU-to-CPU data transfer. Moreover, materializing the free space intervals for all $(m-1)(n-1)$ segment pairs, referred to as cells, for long paths may lead to prohibitive memory consumption, motivating more scalable GPU centric designs.

B. Approximate and Heuristic Methods

When FD is computationally prohibitive, many practical systems adopt approximate similarity measures. A common alternative is the discrete Fréchet distance (DFD) [16], which compares two paths using only their sampled states and the order preserving coupling between samples. While it preserves traversal order, it does not model continuous motion along segments. Thus, it can be computed directly via dynamic programming, avoiding the repeated decider calls required by decision based frameworks. Its value is never smaller than FD and can deviate from FD depending on sampling density and segment lengths of the paths.

Another practical surrogate closely related to FD is VE-FD [15], a relaxation of FD that is guaranteed to be no greater than FD. The VE-FD is defined as the minimum value over all VE paths, where a VE path is a traversal of two paths from the start vertices to the end vertices in which the vertex indices advance monotonically. At each step, one curve advances to its next vertex while the other remains fixed. Each such step is called a VE event and incurs the corresponding vertex-segment distance. The value of a VE path is the maximum vertex-segment distance encountered along the traversal.

IV. APPROACH

A. Overview

To efficiently compute FD between paths on a GPU, we follow the foundational FD framework [10] and accelerate this pipeline end-to-end on a GPU while producing the same FD result. In doing so, we aim to reduce the number of expensive decider calls, organize the data in GPU memory to increase locality and minimize host-device transfers, and efficiently parallelize the reachability propagation. Guided by these considerations, we tighten bounds to reduce the number of decision invocations, redesign data layouts to make free space interval access GPU efficient, and parallelize reachability propagation on the GPU using a wavefront style execution strategy, thereby avoiding the need to transfer free space intervals to CPU. Finally, for long paths where storing the full free space diagram becomes memory prohibitive, we introduce Memory-Saving mode that computes free space intervals on demand and discards them after use.

B. Bounding

Decision based FD computation maintains an upper bound U and a lower bound L on the true distance and iteratively refines them by testing a midpoint threshold $\varepsilon = (L+U)/2$ with the decider and updating L or U accordingly. The tightness of L and U directly impacts performance: loose bounds enlarge the search range and increase the number of expensive decision invocations. Following common practice, we initialize U using a tight bound DFD [16]. In contrast, for L, prior systems [14] often initialize with the maximum of the start vertex distance and end vertex distance (referred to as the simple LB), or the maximum nearest-neighbor distance from the vertices of either path to the other path (referred to as the projective LB). These bounds are computed at low cost but can be far from FD.

Algorithm 1: GPU-parallelized VE-FD computation

Input: Polygonal paths $P = (p_0, \ldots, p_{m-1})$ and $Q = (q_0, \ldots, q_{n-1})$
Output: Lower bound L_{VE} on $d_F(P, Q)$

// $R_{i,j}$ and $T_{i,j}$ are ∞ when $i < 0$ or $j < 0$, except $R_{-1,0} = \max(dist(p_0, q_0), dist(p_{m-1}, q_{n-1}))$
for $k \leftarrow 0$ **to** $m + n - 4$ **do**
  for *each cell* (i, j) *with* $i + j = k$ *on GPU* **do in parallel**
    // Distances from neighbors
    $in \leftarrow \min(T_{i,j-1}, R_{i-1,j})$;
    // $dist(a, (b, c))$ calculates the distance from a vertex a to a line segment (b, c)
    $T_{i,j} \leftarrow \max(in, dist(q_{j+1}, (p_i, p_{i+1})))$;
    $R_{i,j} \leftarrow \max(in, dist(p_{i+1}, (q_j, q_{j+1})))$;
    // Distances to neighbors
    $StoreDistance(i, j, R_{i,j}, T_{i,j})$;
  $Synchronization(all\ threads\ in\ GPU)$;
return $\min(T_{m-2,n-2}, R_{m-2,n-2})$;

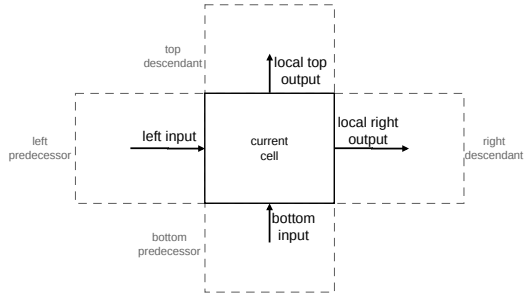


Fig. 2. Per-cell dependency of the VE-FD recurrence. The current cell uses the left and bottom predecessor values and its local vertex-segment distances to compute local outputs and propagate them to its two descendants.

To reduce the number of iterations, we adopt VE-FD [15], which is closer to FD than the two bounds above, as our lower bound. The original VE-FD algorithm runs in $O(mn \log(mn))$ time for two paths with m and n vertices, because it maintains the best candidate path throughout the computation to recover the VE path associated with the distance. Its complexity is comparable to that of FD, making the original algorithm too expensive to use as a practical lower-bound computation. However, because we use VE-FD only as a lower bound and the actual VE path is irrelevant, we can reduce the time complexity of VE-FD to $O(mn)$. Also, as shown in Fig. 2, the recurrence propagates two per-cell values $(top, right)$ using only the left and bottom predecessors and local vertex-segment distances, which enables a wavefront based parallel computation.

Algorithm 1 summarizes our GPU-parallelized $O(mn)$ VE-FD algorithm. An $(m-1) \times (n-1)$ grid is created, and the cells in the grid are processed in anti-diagonal order from $(0,0)$ to $(m{-}2, n{-}2)$, so that all cells on the same anti-diagonal are independent and can be processed in parallel. The final VE-FD is read from the terminal cell of the grid. The correctness follows by induction over anti-diagonals: $T_{i,j}$ and $R_{i,j}$ store the optimal bottleneck values for paths ending at the top and right edges of $cell_{i,j}$, so the terminal value $\min(T_{m-2,n-2}, R_{m-2,n-2})$ equals VE-FD; the full proof is available in our GitHub repository together with the implementation.

As in the original definition of VE-FD [15], every VE path corresponds one-to-one to a monotone path on the grid from $cell_{0,0}$ to $cell_{m-2,n-2}$, where each traversed cell edge represents a VE event that matches a vertex of one curve to a line segment of the other and has weight equal to the corresponding vertex-segment distance.

In addition, the endpoint vertex-vertex pairs (p_0, q_0) and (p_{m-1}, q_{n-1}) are visited by every VE path but are not represented by any cell edge. To account for these endpoint pairs, we incorporate them into the bottleneck by initializing:

$$R_{-1,0} = \max\big(\|p_0 - q_0\|,\ \|p_{m-1} - q_{n-1}\|\big) \tag{3}$$

For each $cell_{i,j}$, $T_{i,j}$ and $R_{i,j}$ are the minimum possible bottleneck values among all grid paths from $cell_{0,0}$ to $cell_{i,j}$ that end at the top and right edges of $cell_{i,j}$, respectively, where the bottleneck value of a path is the maximum edge weight along it. $w_{i,j}^{\text{top}}$ and $w_{i,j}^{\text{right}}$ are the vertex-segment distances associated with the top and right edges of $cell_{i,j}$.

Any path that ends at the top edge of $cell_{i,j}$ must enter $cell_{i,j}$ either from the left via the right edge of $cell_{i-1,j}$ or from below via the top edge of $cell_{i,j-1}$. Therefore, the best achievable bottleneck value upon entering $cell_{i,j}$ is $\min(R_{i-1,j}, T_{i,j-1})$, and after traversing the top or right edge, its bottleneck becomes:

$$T_{i,j} = \max\big(w_{i,j}^{\text{top}},\ \min(T_{i,j-1}, R_{i-1,j})\big). \tag{4}$$

$$R_{i,j} = \max\big(w_{i,j}^{\text{right}},\ \min(T_{i,j-1}, R_{i-1,j})\big). \tag{5}$$

These recurrences are exactly what the algorithm computes in lines 3-6, with boundary conditions initialized from $R_{-1,0}$ at $cell_{0,0}$. Hence, by induction over increasing $i + j$, the algorithm computes the correct $T_{i,j}$ and $R_{i,j}$ for all cells.

Finally, any complete VE path must leave $cell_{m-2,n-2}$ through either its top edge or its right edge, so the optimal bottleneck value is $\min(T_{m-2,n-2}, R_{m-2,n-2})$, which is the algorithm output.

Implementation note. Algorithm 1 uses a synchronized wavefront for clarity, but our implementation is optimized with the techniques described in Section IV-D. DFD follows a wavefront dependency, so the same wavefront parallel engine is also used to compute DFD on the GPU.

C. Interval Computation and GPU Memory Layout

Given a distance threshold ε, the decider requires computing the free space intervals. The computation is fully parallelizable: each vertex-segment evaluation is independent and can be carried out without communication, making it well suited for GPUs. To minimize memory traffic and avoid redundant computation, we map each thread to a unique line segment of one of the paths, so all cells processed by the same thread reuse the segment. Note that the free space intervals computed in this stage are consumed in the same wavefront order as the cells are processed by our GPU reachability propagation in Section IV-D. To improve locality, we arrange the per-cell interval data in the same order, at the cost of additional memory overhead due to padding shorter diagonals to match the longest one.

D. GPU Wavefront Reachability Propagation

Given the free space intervals under threshold ε, reachability propagation determines whether a monotone path exists from the start cell to the end cell. A cell's reachable intervals are the subsets of its free space intervals that are reachable from the start cell by continuously and monotonically traversing the two curves while maintaining a separation of at most ε throughout the traversal. Each cell updates its reachable intervals using its free space intervals and the incoming reachable intervals from its left and bottom predecessors. Based on this dependency, we design an anti-diagonal wavefront GPU-parallelized algorithm for this stage, with an inter-block optimization that enables asynchronous handoff between blocks.

Algorithm 2: Reachability propagation

Input: Per-cell free space intervals I
Output: $feasible$ (whether the end cell is reachable)

```
1  for each thread do
2      LExch, RExch ← GetExchange(thread_id);
3      botR ← unreachable;
4      if thread_id=0 then
5          botR ← (0, ∞);
6  for each block b on GPU do in parallel
7      for k ← 0 to m + n − 4 do
8          for each cell (i, j) (if it exists) assigned to b
             on diagonal k do in parallel
               // (1) Read leftR from buffer
9              leftR ← unreachable;
10             if HasLeftNeighbor(b) then
11                 Wait until buffer is ready to be read;
12                 leftR ← Read(LExch);
               // (2) Reachability Propagation
13             topR, rightR ←
                 Reachability(leftR, botR, I_{i,j});
               // topR advances within the column
14             botR ← topR;
               // (3) Write rightR to buffer
15             Write(RExch, rightR);
16         Synchronization(All Threads in b);
17 return true if either leftR or botR of cell_{m−2,n−2}
     is reachable, otherwise false;
```

Algorithm 2 outlines our algorithm. The reachability propagation of a cell (line 13) follows the FD reachability propagation rules in [10]. All cells are processed in anti-diagonal wavefronts, and we partition each wavefront into column strips and assign one thread per column so that $topR$ values remain thread local while only $rightR$ values require inter-column communication (i.e., the $rightR$ of cell (i, j) is passed to the $leftR$ of cell $(i+1, j)$). Within each block, neighboring columns exchange reachable intervals through shared memory: thread t produces the right reachable interval needed by thread t+1. Only the right reachable interval produced by the last thread of a block therefore requires a cross-block exchange via global memory, because it is consumed by the next block. To enable asynchronous handoff between blocks, we implement this inter-block communication with a global memory buffer: block b–1 writes its reachable interval to the buffer, and block b waits for the write to complete before reading from the buffer. The buffer can hold multiple reachable intervals when block $b-1$ runs ahead of block b. In line 2, $RExch$ and $LExch$ denote exchange slots whose storage depends on the communication scope: they reside in shared memory for intra-block handoff, and are mapped to the buffer for the inter-block handoff.

In our implementation, in order to sustain high occupancy and minimize inter-block access, we launch one block per Streaming Multiprocessor (SM), set the block size to m/B threads with B blocks, and assign one column per thread. If m/B exceeds the maximum available block size s, we use a global strided mapping: the thread g processes columns $g + k * sB$ for $k = 0, 1, 2, \ldots$. In addition, we allocate the buffer as large as permitted by the remaining GPU memory budget to reduce the likelihood of stalling due to saturation.

E. Memory-bounded Computation for Long Paths

The interval computation in Section IV-C produces per-cell free space intervals for the entire $(m-1) \times (n-1)$ free space diagram. For each cell, we store four free space interval endpoints (top and right, two each), leading to a memory footprint on the order of

$$\mathcal{M}_{\text{interval}} \approx 4\,(m-1)(n-1) \cdot \text{sizeof(data type)}. \quad (6)$$

We refer to this implementation as Performance mode.

For long paths, however, $\mathcal{M}_{\text{interval}}$ can become prohibitive and cannot fit in the GPU memory, making full materialization of the free space diagram infeasible. To handle this case, we introduce Memory-Saving mode that avoids storing the free space intervals. Instead of precomputing all intervals, we fuse interval computation with reachability propagation: whenever Algorithm 2 updates a cell during wavefront propagation (line 13), we first compute the required local interval information $I_{i,j}$ on demand and immediately apply it to update top and right reachable intervals, after which $I_{i,j}$ is discarded. This mode eliminates the need to materialize all the free space intervals.

However, such fusion is not always beneficial: it may increase the waiting time on intra-block synchronization. Because free space interval computation has non-uniform cost across cells, the synchronization in Memory-Saving mode makes faster threads idle until the slowest threads finish for each wavefront. Moreover, fully materializing all the free space intervals exposes massive parallelism, because interval computation has $(m-1)(n-1)$ independent tasks. By contrast, reachability propagation exposes only wavefront level parallelism bounded by $\min(m-1, n-1)$. As a result, on-demand evaluation can underutilize the GPU when paths are short. Thus, we only apply this fusion when the free space interval data cannot fit in the GPU memory.

V. EVALUATION

A. Experimental Setup

All experiments were conducted on a workstation equipped with a 12th Gen Intel Core i9-12900 CPU with 16 cores (up to 5.1 GHz), 62 GB of RAM, and an NVIDIA GeForce RTX 3090 GPU with 24 GB of VRAM.

We compare our method against the official implementation of the best existing GPU accelerated FD method [14], Fréchet-GPU, and its multicore CPU variant proposed in the same work, Fréchet-Multicore. For fair and stable timing, we apply a minor code fix to the bounding stage of Fréchet-Multicore by adding the same short input parallelization guard used elsewhere in the released code base. This guard

TABLE I
PATH LENGTH (NUMBER OF VERTICES) STATISTICS FOR EACH DATASET.

Dataset	Min Length	Max Length	Average Length
DROID100	2	99	70
DROID1000	102	912	308
DROID3000	1000	2908	1385
DROID5000	3007	4498	3249
RobotCarLow	1189	9741	4085
RobotCarHigh	11930	51841	22890
UZH-FPV	8800	42850	19344
NCLT	150971	997428	712868

only reduces overhead on very small paths (< 1000 vertices). Although all methods can support any precision, the datasets are mostly stored in single precision. We therefore run all methods in double precision for numerical robustness, and terminate the search once the gap between the lower and upper bounds is below machine epsilon for the float type, targeting single precision accuracy, since higher precision would only increase the number of iterations.

To evaluate the methods, we use the Cartesian position path traces extracted from four datasets spanning representative robotics application domains, robot platforms, and operating environments. **DROID** [17] consists of large scale indoor robot manipulation demonstrations and provides end-effector trajectories representative of manipulation settings studied in imitation learning [18] and task space motion planning and control [4]. **Oxford RobotCar** [19], [20] and **NCLT** [21] provide long real world ground robot trajectories with rich onboard sensing and reference position streams. They are widely used in SLAM [22] and odometry evaluation [23]. In addition to these ground robot datasets, **UZH-FPV** [24] provides aerial trajectories from an FPV racing quadrotor with aggressive and highly dynamic motion, making it representative of drone-racing odometry settings [25].

Because DROID and Oxford RobotCar vary widely in path length, we further partition them into multiple datasets. DROID is partitioned into DROID100, DROID1000, DROID3000, and DROID5000. Oxford RobotCar is partitioned into RobotCarLow and RobotCarHigh. Detailed dataset statistics are reported in Table I.

B. Runtime Performance

To evaluate runtime across different path lengths, we randomly sampled 32 distinct path pairs per dataset. Table II reports the breakdown of FD computation time for all three methods. The total time of the fastest method is highlighted in bold for each dataset. Fréchet-GPU fails on NCLT due to running out of GPU memory. Fréchet-Multicore does not finish within 48 hours for UZH-FPV, RobotCarHigh, and NCLT. Also, for UZH-FPV, RobotCarHigh, and NCLT, due to memory limitations, **RobotFD** was run in Memory-Saving mode, so the runtime is not broken down into interval computation and reachability propagation. Fréchet-Multicore does not move data to GPU, while both **RobotFD** and Fréchet-Multicore do not move data to CPU, so the times for the corresponding stages are not applicable for these methods.

Overall, **RobotFD** is comparable to or faster than Fréchet-GPU and Fréchet-Multicore on all datasets except DROID100. Across the datasets, **RobotFD** achieves a speedup of up to $39\times$ in total time. For datasets with longer paths, GPU parallelization enables **RobotFD** to achieve greater speedup. On the extremely long NCLT path traces, where both Fréchet-GPU and Fréchet-Multicore cannot finish, **RobotFD** finishes within one hour, enabling exact distance evaluation in a regime previously impractical.

The only exception is DROID100, where Fréchet-GPU is the fastest. This behavior is expected for the small paths (< 100 vertices): the additional cost of computing a tighter yet more complex lower bound exceeds the decider cost it saves, because the decider work is relatively light for short paths. Moreover, reachability propagation offers limited parallelism at this scale, so kernel launches and synchronization overheads can dominate and negate any benefit. Therefore, for paths shorter than 100 vertices ($m \times n < 10000$), we recommend the strategy proposed by Fréchet-GPU. **RobotFD** automatically applies this strategy in our implementation.

In absolute terms, on most **DROID** subsets, **RobotFD** sustains a throughput of tens of path pairs per second on average, which is compatible with online local replanning pipelines that commonly evaluate candidate motion segments at around $10\,\text{Hz}$ [26], [27]. Accordingly, FD based scoring becomes practical in high volume similarity queries for imitation learning [28] and in offline quantification of manipulation planning and control [5]. For **Oxford RobotCar** and **UZH-FPV**, our per-pair runtime ranges from subsecond to several seconds, making RobotFD suitable for their common use as offline benchmarks in SLAM and odometry evaluations [29]. Finally, **NCLT** contains extremely long path traces, and **RobotFD** can take about one hour per pair on this dataset. It is commonly used for offline long horizon benchmarking of SLAM on a limited set of pairs, and our runtime is practical for this use case. Taken together, these results indicate that **RobotFD** delivers robust end-to-end gains across realistic path traces, from online short horizon scoring to offline long horizon benchmarking, making FD practically deployable at scale.

Table II also breaks down the total time into moving data to GPU memory, bounding, interval computation, moving data to CPU memory, and reachability propagation. The time spent moving data to GPU memory is always negligible or not applicable. In the **bounding** stage, **RobotFD** uses a more complex bound than prior baselines. As a result, for paths with fewer than 1,000 vertices, **RobotFD** is slower than the baseline methods in this stage. For longer paths, taking advantage of GPU parallelization, **RobotFD** is often faster than the baseline methods in this stage. In the **interval computation** stage, **RobotFD** consistently outperforms both baselines due to fewer iterations, the GPU friendly memory layout, and the strategy that avoids redundant computation. In the **moving data to CPU** stage, Fréchet-GPU accounts for a significant portion of the runtime on long paths with more than 1,000 vertices, while other methods do not need to move

TABLE II

COMPARISON OF THE RUNTIME AND ITS DECOMPOSITION IN MILLISECONDS ("–" INDICATES THAT THE VALUE IS NOT APPLICABLE)

Dataset	Method	Move Data to GPU	Bounding	Interval Computation	Move Data to CPU	Reachability Propagation	Total	$Speedup(\frac{\text{Baseline}}{\text{RobotFD}})$
DROID100	RobotFD	0.02	13	0.17	–	2.3	15	–
	Fréchet-GPU	0.05	0.13	6.3	0.36	0.18	**7**	0.46×
	Fréchet-Multicore	–	1.1	13	–	1.6	16	1.0×
DROID1000	RobotFD	0.02	14	0.42	–	7.5	**22**	–
	Fréchet-GPU	0.05	2.6	22	4.7	8.8	38	1.7×
	Fréchet-Multicore	–	9.0	28	–	12	48	2.2×
DROID3000	RobotFD	0.03	18	16	–	40	**74**	–
	Fréchet-GPU	0.06	59	351	90	288	787	11×
	Fréchet-Multicore	–	129	605	–	351	1085	15×
DROID5000	RobotFD	0.04	30	95	–	93	**218**	–
	Fréchet-GPU	0.07	407	1658	442	1774	4281	20×
	Fréchet-Multicore	–	152	3059	–	3711	6922	32×
RobotCarLow	RobotFD	0.05	33	257	–	144	**434**	–
	Fréchet-GPU	0.07	873	4007	1054	4691	10626	24×
	Fréchet-Multicore	–	409	6008	–	9588	16004	37×
RobotCarHigh	RobotFD	0.45	286	8495			**8781**	–
	Fréchet-GPU	0.37	18776	101959	28647	173583	322966	37×
	Fréchet-Multicore	–						–
UZH-FPV	RobotFD	1.0	197	4702			**4900**	–
	Fréchet-GPU	0.50	15363	47360	13216	113670	189610	39×
	Fréchet-Multicore	–						–
NCLT	RobotFD	2.4	96177	3234989			**3331168**	–
	Fréchet-GPU	–						–
	Fréchet-Multicore	–						–

TABLE III

NUMBER OF DECIDER INVOCATIONS (ITERATIONS)

Dataset	RobotFD	Fréchet-GPU	Fréchet-Multicore
DROID100	8.4	11.8	10.5
DROID1000	9.9	13.8	16.8
DROID3000	14.3	19.1	20.6
DROID5000	13.5	15.7	20.0
RobotCarLow	15.6	17.8	18.1

data to CPU. Finally, in the **reachability propagation** stage, GPU parallelism yields additional gains over Fréchet-GPU and Fréchet-Multicore (except on very short paths where parallelism is insufficient). As path length grows, parallelism increases and the speedup from this stage becomes more pronounced in **RobotFD**. **Overall**, the runtime breakdown shifts from being dominated by bounding and interval computation on short paths to being dominated by interval computation and reachability propagation on longer paths.

C. Analysis of Bounds

We compare the lower bound used by **RobotFD** against the simple LB used by Fréchet-GPU and the projective LB used by Fréchet-Multicore. To make the comparison controlled and fair, we only report results on datasets on which all three methods complete successfully. For each dataset, we randomly sample 32 distinct path pairs, run the three methods, and record the average number of decider invocations. Because the number of decider invocations depends only on the bounds, this comparison directly shows the tightness of bounds.

As shown in Table III, **RobotFD** consistently invokes the decider about 10–30% fewer times than the better-performing baseline, indicating that VE-FD provides a substantially tighter lower bound and can closely track the true FD in practice. This reduction directly translates to fewer calls to the decision procedure, which is computationally expensive.

D. Tradeoff of Memory-Saving Mode

To quantify the impact of Memory-Saving mode, we focus on datasets where the free space intervals fit in GPU memory. For each dataset, we randomly sample 32 distinct path pairs and compare Performance mode vs. Memory-Saving mode in (i) end-to-end runtime and (ii) GPU memory consumption excluding the buffer. Since we size the buffer to fill the remaining GPU memory, including it would make the memory footprint uninformative.

Fig. 3 reports the GPU memory footprint and runtime for the two modes. We observe that the memory usage of Performance mode grows super-linearly with path length. When the length approaches 10,000 vertices, the additional state can consume nearly 1 GB of GPU memory. In contrast, Memory-Saving mode keeps the extra memory overhead below 1 MB. This benefit comes at a performance cost: reduced parallelism and additional synchronization requirements slow down the overall execution by about 13–69% across the evaluated datasets. Consequently, Memory-Saving mode is best suited for extremely long paths where Performance mode becomes memory limited, while Performance mode remains preferable when memory is sufficient.

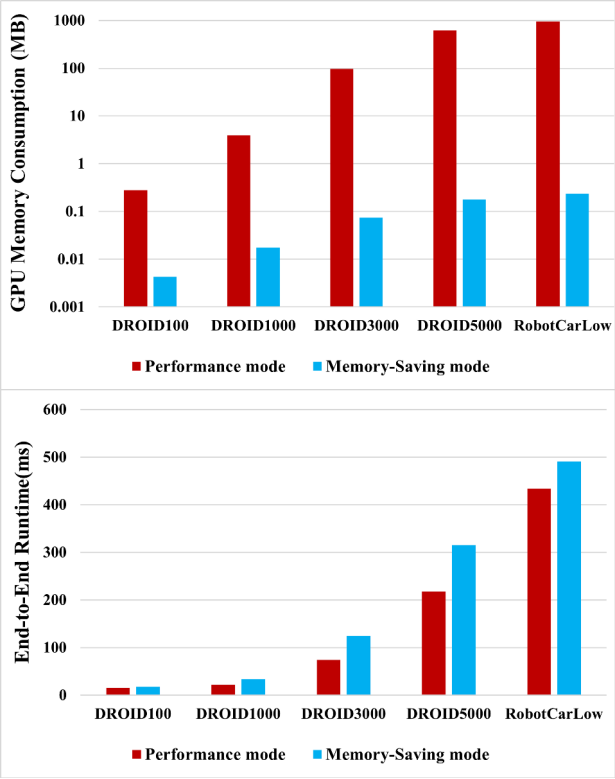


Fig. 3. Comparison of Performance mode and Memory-Saving mode in terms of GPU memory consumption excluding the buffer (top) and runtime (bottom). Because the memory-consumption values span several orders of magnitude, the top plot uses a logarithmic scale.

VI. CONCLUSIONS

RobotFD makes exact FD computation practical at scale for robotic path scoring and comparison. Specifically, **RobotFD** integrates tighter GPU computed bounds with a fully GPU resident wavefront decider and Memory-Saving mode, enabling high throughput evaluation on typical path lengths while maintaining robust performance on extremely long paths under strict memory constraints. Across real world datasets of varying sizes, **RobotFD** outperforms both the state-of-the-art GPU baseline and the corresponding multicore implementation in most scenarios, achieving up to 39× end-to-end speedup and extending FD computation into regimes where prior methods become impractical. Overall, **RobotFD** makes FD viable both for online local replanning with small candidate sets and for large scale offline long horizon benchmarking in practical robotics workloads.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their constructive comments and suggestions. This work is supported in part by the U.S. National Science Foundation under Grants CCF2005884, CCF-2210753, CCF2312507, and OAC-2310510.

REFERENCES

[1] M. R. Montero et al., "Learning multi-reference frame skills from demonstration with task-parameterized Gaussian processes," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2024, pp. 2832–2839.

[2] C. Innes et al., "Elaborating on learned demonstrations with temporal logic specifications," in Proc. Robot. Sci. Syst. (RSS), 2020.

[3] B. El Khadir et al., "Teleoperator imitation with continuous-time safety," in Proc. Robot. Sci. Syst. (RSS), 2019.

[4] R. Holladay et al., "Minimizing task-space Fréchet error via efficient incremental graph search," IEEE Robot. Autom. Lett., vol. 4, no. 2, pp. 1999–2006, 2019.

[5] R. M. Holladay et al., "Distance metrics and algorithms for task space path optimization," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2016, pp. 5533–5540.

[6] S. Niyaz et al., "Optimizing motion-planning problem setup via bounded evaluation with application to following surgical trajectories," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2019, pp. 1355–1362.

[7] J. Sturm et al., "A benchmark for the evaluation of RGB-D SLAM systems," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2012, pp. 573–580.

[8] Z. Zhang et al., "A tutorial on quantitative trajectory evaluation for visual(-inertial) odometry," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2018, pp. 7244–7251.

[9] D. Schubert et al., "The TUM VI benchmark for evaluating visual-inertial odometry," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2018, pp. 1680–1687.

[10] H. Alt et al., "Computing the Fréchet distance between two polygonal curves," Int. J. Comput. Geom. Appl., vol. 5, no. 1–2, pp. 75–91, 1995.

[11] K. Buchin et al., "Four soviets walk the dog: Improved bounds for computing the Fréchet distance," Discrete Comput. Geom., vol. 58, pp. 180–216, 2017.

[12] K. Solovey et al., "Efficient sampling-based bottleneck pathfinding over cost maps," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2017, pp. 2003–2009.

[13] S. Niyaz et al., "Following surgical trajectories with concentric tube robots via nearest-neighbor graphs," in Proc. Int. Symp. Exp. Robot. (ISER), 2018, pp. 3–13.

[14] S. Meintrup et al., "Random projections and sampling algorithms for clustering of high-dimensional polygonal curves," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), vol. 32, 2019.

[15] S. Har-Peled et al., "The Fréchet distance unleashed: Approximating a dog with a frog," in Proc. Int. Symp. Comput. Geom. (SoCG), 2025, pp. 54:1–54:13.

[16] T. Eiter et al., "Computing discrete Fréchet distance," Tech. Rep. CD-TR 94/64, 1994.

[17] A. Khazatsky et al., "DROID: A large-scale in-the-wild robot manipulation dataset," in Proc. Robot. Sci. Syst. (RSS), 2024.

[18] F. Lin et al., "Data scaling laws in imitation learning for robotic manipulation," in Proc. Int. Conf. Learn. Represent. (ICLR), 2025.

[19] W. Maddern et al., "1 year, 1000 km: The Oxford RobotCar dataset," Int. J. Robot. Res., vol. 36, no. 1, pp. 3–15, 2017.

[20] W. Maddern et al., "Real-time kinematic ground truth for the Oxford RobotCar dataset," arXiv:2002.10152, 2020.

[21] N. Carlevaris-Bianco et al., "University of Michigan North Campus long-term vision and lidar dataset," Int. J. Robot. Res., vol. 35, no. 9, pp. 1023–1035, 2016.

[22] G. Kim et al., "1-day learning, 1-year localization: Long-term LiDAR localization using scan context image," IEEE Robot. Autom. Lett., vol. 4, no. 2, pp. 1948–1955, 2019.

[23] Y. Wei et al., "BEV-ODOM: Reducing scale drift in monocular visual odometry with BEV representation," arXiv:2411.10195, 2024.

[24] J. Delmerico et al., "Are we ready for autonomous drone racing? The UZH-FPV drone racing dataset," in Proc. IEEE Int. Conf. Robot. Autom. (ICRA), 2019, pp. 6713–6719.

[25] G. Cioffi et al., "Learned inertial odometry for autonomous drone racing," IEEE Robot. Autom. Lett., vol. 8, no. 5, pp. 2684–2691, 2023.

[26] R. A. Knepper et al., "Hierarchical planning architectures for mobile manipulation tasks in indoor environments," in Proc. IEEE Int. Conf. Robot. Autom. (ICRA), 2010, pp. 1985–1990.

[27] H. H. Nguyen et al., "Language-driven closed-loop grasping with model-predictive trajectory optimization," Mechatronics, vol. 109, Art. no. 103335, 2025.

[28] O. Kwon et al., "RT-Cache: Training-free retrieval for real-time manipulation," in Proc. IEEE-RAS Int. Conf. Humanoid Robots (Humanoids), 2025, pp. 1–8.

[29] T. Scargill et al., "SiTAR: Situated trajectory analysis for in-the-wild pose error estimation," in Proc. IEEE Int. Symp. Mixed Augment. Reality (ISMAR), 2023.