2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3195-0602-3/26/$31.00 ©2026 IEEE | DOI: 10.1109/IPDPS65963.2026.00043

DSTREE: Data-Driven Synchronous Traversals for Decision Forests on GPUs

Bi Zeng*, Zhenlin Wu*, Hongyuan Liu, and Xinyu Chen†
The Hong Kong University of Science and Technology (Guangzhou)
{bzeng403, zwu065}@connect.hkust-gz.edu.cn, {hongyuanliu, xinyuchen}@hkust-gz.edu.cn

***Abstract*—Decision forests are widely adopted for their strong accuracy and interpretability. Their inherently parallel structure, where trees and queries can be evaluated independently, makes them a natural fit for GPUs with massive thread-level parallelism and high memory bandwidth. However, conventional GPU implementations typically bind independent tasks statically to GPU threads, leading to irregular control flow and scattered memory access patterns. This paper proposes DSTREE, a dynamic, data-driven inference framework that replaces static execution with a level-synchronous worklist traversal. DSTREE enables GPU threads to cooperatively fetch and process active nodes, improving warp utilization and memory access efficiency. Building on top of the worklist, we introduce privatized synchronous traversal to reduce synchronization overhead and a worklist reordering mechanism to improve memory coalescing and data locality. Evaluation across six real-world datasets demonstrates that our method delivers up to 5.3× (3.9× on average) over the state-of-the-art RAPIDS Forest Inference Library (FIL), and up to 3.5× (1.8× on average) speedup over Tahoe.**

I. INTRODUCTION

Decision forests, encompassing models such as Random Forests [1]–[5] and Gradient-Boosted Tree ensembles [6], [7], have become a cornerstone of machine-learning systems for tabular data [8]–[10]. They offer strong predictive accuracy, robustness to noise and heterogeneous features, and inherent interpretability through explicit decision paths [11]. Consequently, decision forests are widely deployed in domains such as medicine prediction [12], [13], medical diagnosis [14], data mining [15], database management [16], and recommendation systems [17]–[19]. Despite their popularity, the inference stage, which evaluates thousands of trees across large batches of queries, remains a major performance bottleneck [20], [21].

Modern Graphics Processing Units (GPUs) appear to be a natural fit for accelerating large-scale decision forest inference [22]–[29]. Each tree within an ensemble and each query can be evaluated independently, exposing abundant parallelism [30]. Modern GPUs provide massive thread-level parallelism, high memory bandwidth, and wide SIMD-style execution, which in principle align well with the structure of decision-forest workloads. Frameworks such as NVIDIA RAPIDS [30] have demonstrated the potential of GPUs for decision forests.

However, in practice, GPU-based decision-forest inference remains far from efficient. The root-to-leaf traversal in each tree introduces irregular control flow, since different queries follow distinct paths, and scattered memory access, because tree nodes are sparsely distributed in memory. These characteristics lead to two fundamental inefficiencies: 1) **Thread underutilization** – divergent traversal paths cause threads within a warp to finish at different times, leaving many lanes idle. 2) **Uncoalesced memory accesses** – threads in the same warp often access non-contiguous nodes, preventing memory transactions from being coalesced and drastically reducing effective bandwidth utilization.

Existing GPU inference engines, such as RAPIDS Forest Inference Library (FIL) [30] and Tahoe [31], mitigate these issues through static scheduling and shallow-layer optimizations. FIL assigns each thread to a fixed (query, tree) pair throughout traversal, while Tahoe pads trees into near-perfect binary structures or caches top layers in on-chip memory to improve access regularity. Although effective for small, shallow forests, these designs degrade rapidly as tree depth and size increase. We observe that thread utilization is often below 50% during the execution and drops significantly as depth increases (Figure 2), while the memory-coalescing ratio falls below 10% when the trees are deep.

These observations motivate a fundamentally different design. To this end, our key insight is that instead of assigning a fixed traversal path to each thread, we adopt a dynamic, data-driven execution model that continuously redistributes work and reorganizes memory accesses. Building on this insight, we propose DSTREE, a worklist-centric inference framework that restructures decision-forest execution to better utilize GPU hardware. At each level, GPU threads cooperatively fetch and process tasks from this worklist, improving thread utilization even when traversal paths diverge. To balance synchronization overhead and thread utilization, DSTREE allows threads to advance several traversal steps before global synchronization. To enhance memory efficiency, DSTREE periodically reorders tasks within the worklist so that threads in the same warp access spatially adjacent nodes. In summary, we make the following contributions:

- We identify that the root cause of performance inefficiency in GPU-based decision forest inference lies in static scheduling, which leads to low thread utilization and poor memory coalescing.
- We propose DSTREE, a worklist-centric GPU-based decision forest engine with a dynamic scheduling strategy that reassigns tasks at runtime to reduce idle threads and improve

*Both authors contributed equally to this work.
†Corresponding author.

1530-2075/26/$31.00 ©2026 IEEE
DOI 10.1109/IPDPS65963.2026.00043

thread utilization.

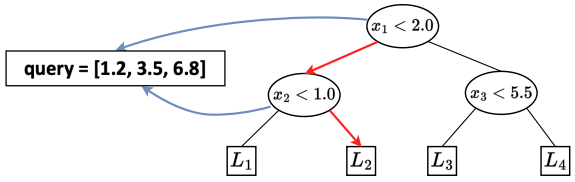
- We introduce a task reordering mechanism that enhances warp-level spatial locality and memory coalescing.
- Our evaluation demonstrates that DSTree achieves up to 5.3× (3.9× on average), and up to 3.5× (1.8× on average) speedup over RAPIDS FIL, and Tahoe, respectively.

II. Background

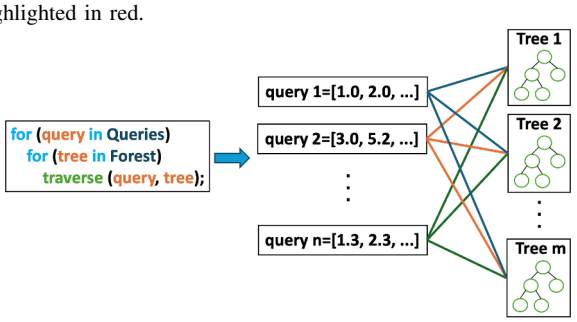
A. Decision Tree and Decision Forest

A decision tree is a hierarchical model that recursively partitions the input feature space based on learned thresholds. As illustrated in Figure 1 (a), single decision tree inference begins at the root node. At each internal node, the algorithm compares the query feature value indicated by the stored `feature_id` against the threshold (e.g. $x_1 < 2.0$ in the example). Based on the comparison result, the traversal proceeds to either the left or the right child node. This process repeats until a leaf node is reached, which provides the prediction value.

However, a single tree often exhibits high variance and limited generalization capability. To overcome this, decision forests aggregate the predictions of multiple trees, typically through averaging (for regression) or majority voting (for classification). As illustrated in Figure 1 (b), given a forest with M trees and a batch of N queries, inference requires performing $N \times M$ independent root-to-leaf traversals. The resulting $N \times M$ traversals can be mapped to GPUs by parallelizing over queries and over trees simultaneously.



(a) Single decision tree inference workflow with actual execution path highlighted in red.



(b) Decision forest inference workflow.

Fig. 1: Decision Tree and Decision Forest Inference.

B. Decision Forest Inference on GPUs

Accelerating decision-forest inference on GPUs requires careful organization of both data layout and execution strategy. Existing frameworks [30], [31] employ several tree-storage formats to optimize different aspects of performance. The depth-first (DFS) layout [32] arranges nodes along root-to-leaf paths, improving per-query locality since each traversal accesses consecutive memory regions. Conversely, the breadth-first (BFS) layout [33] stores nodes level by level, benefiting level-synchronous processing as adjacent threads access similar tree depths simultaneously. A third approach, the interleaved layout, alternates nodes from multiple trees to improve cross-tree parallelism and global-memory coalescing. The interleaved format is commonly used in high-performance libraries such as Tahoe [31] and RAPIDS FIL [30].

In addition to storage layout, inference performance heavily depends on the execution model. The RAPIDS Forest Inference Library adopts a GPU-specific "shared-data" parallel algorithm. In this model, each thread block is responsible for processing one query at a time. The query's feature vector is first loaded into shared memory, and the trees of the ensemble are distributed among threads in a round-robin fashion. Each thread independently traverses its assigned subset of trees to compute partial predictions, which are then aggregated via a block-wide reduction to form the final ensemble output.

III. Motivation

Despite the apparent parallelism of decision forest inference, existing GPU implementations suffer from low utilization and poor scalability. Profiling results from RAPIDS FIL reveal two dominant bottlenecks: thread underutilization caused by static scheduling, and uncoalesced memory access caused by irregular traversal patterns. These issues worsen as the forest depth or query batch size increases.

Observation 1.

Static task assignment in conventional GPU inference engines leads to severe thread underutilization during tree traversal.

In the static task assignment used by FIL, each thread is assigned a fixed subset of tree-traversal tasks for a given query, and this assignment remains unchanged throughout the traversal. Since different queries take distinct paths and reach leaf nodes at varying depths, threads within the same warp complete their tasks at different times. Once a thread finishes early, it remains idle while others continue processing, leading to significant warp divergence and reduced SM occupancy. As the traversal proceeds deeper, this imbalance compounds—threads working on longer paths dominate execution while the rest of the warp stalls. To quantify this effect, we define **Warp Utilization** as the fraction of active threads in a warp at a certain time. This can be defined as

$$U_i = \frac{Active\_Threads_i}{Total\_Threads_i} \quad (1)$$

Here, *Total_Threads* is the number of threads per warp (32 on NVIDIA GPUs). The overall thread utilization is then measured as the geometric mean across all warps:

$$\textit{Overall Thread Utilization} = \left(\prod_{i=1}^{N} U_i\right)^{\frac{1}{N}} \quad (2)$$

We measure thread utilization over all warps and depths during traversal execution. Figure 2 presents results from RAPIDS FIL, showing that utilization declines as tree depth increases, with a sharp drop between depths 20 and 40.

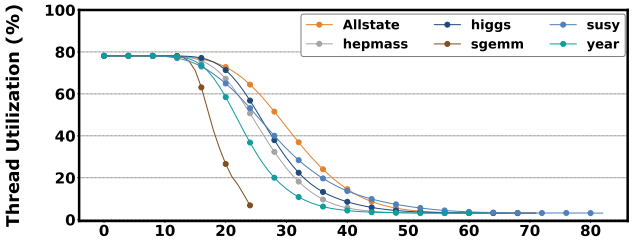


Fig. 2: Thread utilization of different tree depths across different benchmarks for RAPIDS FIL [30] during decision forest inference.

Observation 2.

Static task assignment leads to severely random memory access patterns among threads within the same warp, degrading GPU memory bandwidth utilization.

Even when all threads in a warp remain active, their memory access patterns are highly irregular. Each thread independently fetches tree nodes from distinct memory regions according to its traversal path, and these nodes are typically scattered throughout device memory. As a result, the memory requests issued by the warp cannot be merged into contiguous transactions. GPUs rely on coalesced memory access to combine the requests of adjacent threads into a single memory transaction. When the accessed addresses are far apart, the accesses become uncoalesced, leading to redundant transactions and significantly lower effective memory-bandwidth utilization.

We quantify this effect by measuring the distance between addresses accessed by adjacent threads within the same warp at different depths. Figure 3 reports the fraction of adjacent-thread accesses that fall within a range of memory transactions (128 bytes). The results show that at very shallow depths (depth ≤ 2), nearly all thread accesses remain within 128 bytes. However, as the forest depth increases, this fraction drops sharply. Beyond depth 15, for all datasets, less than 10% of adjacent-thread accesses fall within the 128-byte boundary.

In summary, static task assignment amplifies two key inefficiencies: low thread utilization from varying traversal lengths and poor memory efficiency from random node accesses. These issues limit GPU-based decision-forest inference performance. To address this, we propose a dynamic execution

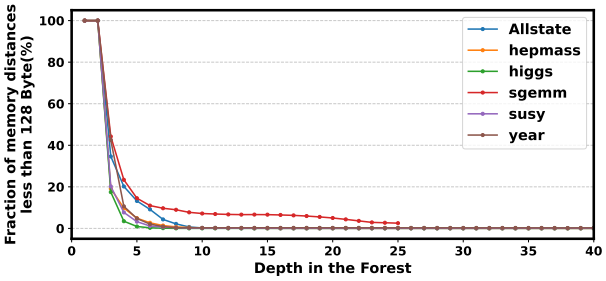


Fig. 3: Fraction of memory accesses that fall within 128 bytes (i.e., a memory transaction size) in a certain depth across different benchmarks for RAPIDS FIL [30].

model to redistribute traversal tasks and optimize memory access patterns. This inspired the design of DSTREE, a dynamic framework that replaces static scheduling with cooperative task fetching. DSTREE ensures that all active threads participate in each traversal level, balancing workload dynamically, and introduces task-reordering mechanisms to coalesce memory transactions. Together, these ideas enable efficient, scalable inference across diverse forest structures.

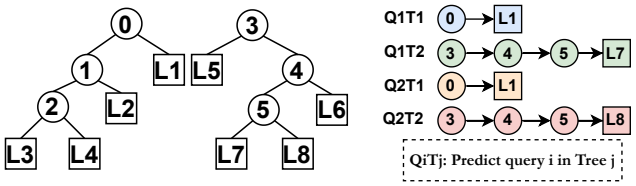
IV. DSTREE DESIGN

In this section, we present DSTREE, a framework that adopts a synchronous traversal strategy to address the issue of thread underutilization.

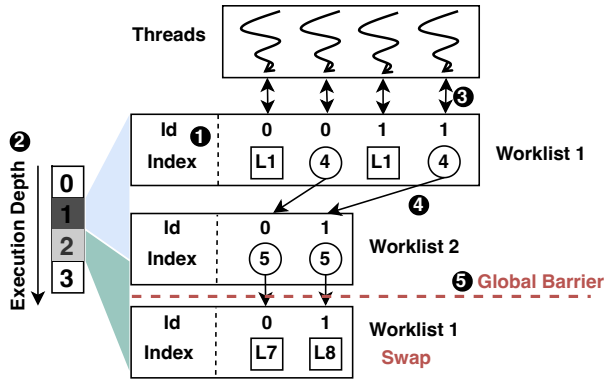
A. Key Insight

Figure 4 illustrates the core idea behind DSTREE's synchronous worklist-based traversal. A decision forest (Figure 4a) consists of multiple trees evaluated across many queries, each following its own path (Figure 4b). Instead of binding each thread to a fixed (query, tree) pair, DSTREE organizes all active traversal tasks into a worklist indexed by tree depth (Figure 4c). At each level, all threads cooperatively fetch and process the active nodes from the current worklist, ensuring that traversal across different trees proceeds in lockstep. Once a level is completed, the updated child nodes are appended to the next-level worklist, followed by a global barrier that synchronizes all threads before advancing.

Figure 5 illustrates how the worklist reduces thread underutilization. In Figure 5a, static task assignment leads to thread underutilization, since thread 1, 3 terminate early and remain idle while others continue to traverse deeper. In Figure 5b, the worklist dynamically collects unfinished tasks and reassigns them to new warps for continued execution. As a result, only a single warp is needed after the third step. In practice, many warps are launched concurrently in a GPU kernel. Therefore, the saved time (the gray regions) can be used to schedule other warps, thereby reducing pipeline bubbles and improving overall GPU throughput.



(a) Random Forest (b) Paths of Queries



(c) Worklist Execution Framework

Fig. 4: (a) Constructed decision forest for the example. (b) Example of paths for queries where `QiTj` indicates predicting query i in Tree j. (c) Worklist-based execution framework.

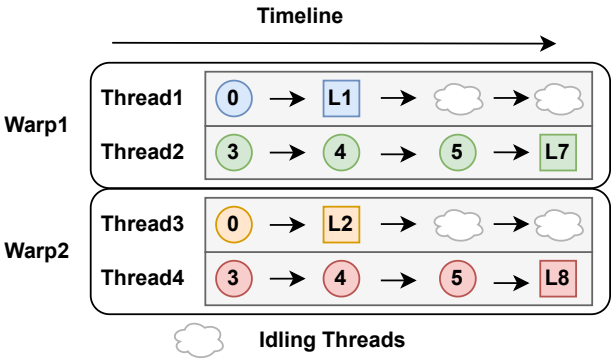
Algorithm 1: Synchronous Traversal (Host Side)

Input: Worklist, Next_Worklist, Forest, Data, Output
1 **while** *Worklist* $\neq \varnothing$ **do**
2 Launch kernel `one_step_traverse` (Worklist, Next_Worklist, Forest, Data, Output);
3 Global_Barrier();
4 Swap(Worklist, Next_Worklist);
5 **end**

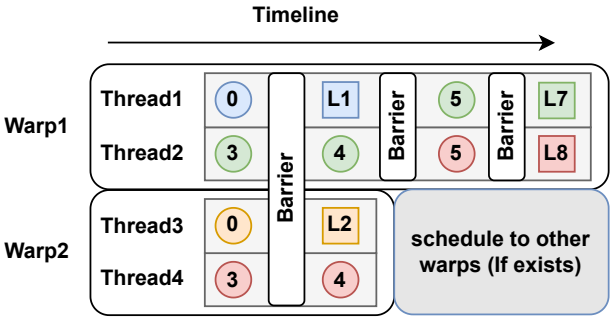
B. Worklist Creation

As discussed in Section II, given a decision forest with m trees and n input queries, the inference process involves performing $n \times m$ independent tree traversals. Therefore, we call each traversal a task. The state of a task is represented as a pair **(Id, Index)** ❶, where the **Id** identifies the input query being processed, and the **Index** denotes the current position of the traversal within the forest. In the worklist, we make all tasks reside at the same tree depth. For example, as shown in the Figure 4c ❷, all tasks in the *current worklist* have reached nodes at depth 1 in the forest.

Algorithm 1 outlines the design on the host side. The algorithm begins with a loop that continues until the worklist becomes empty, ensuring that all unfinished query–tree tasks are eventually completed (line 1). In each iteration, we launch `one_step_traverse` to advance every task by exactly one step and write unfinished tasks into `Next_Worklist` (line 2). After the kernel terminates, `Global_Barrier()` enforces level-synchronous execution by ensuring that all tasks in the current



(a) Execution flow of FIL.



(b) Execution flow with worklist technique.

Fig. 5: Execution Flow of FIL and DSTree

worklist complete the same-depth step before the next iteration (line 3). The host then swaps the current and next worklists, preparing for the subsequent iteration (line 4).

C. Synchronous Traversals

During the processing stage, we launch a kernel where each thread is assigned to execute a task traversal ❸. If the task has already reached a leaf node (represented as rectangles), the thread performs an `atomicAdd` operation to accumulate the result for the corresponding query ID. Otherwise, if the task is still at an internal node (represented as circles), the thread takes a step based on the task's state (id, index). The thread calculates the next position, updates the task index to the new position accordingly, and pushes the updated task into the *next worklist* ❹. To ensure that all tasks in the worklist remain at the same depth within the forest, A global barrier that synchronizes all threads after the current worklist is needed. Once synchronization completes, the *next worklist* will become the new *current worklist*, and the traversal process continues by launching a new kernel ❺. This process repeats until all tasks have reached leaf nodes and the computation is complete.

Algorithm 2 illustrates the device-side process of synchronous worklist traversal. Each GPU thread is assigned one unfinished task from the Worklist (line 2) and advances it by a single traversal step through the decision tree (line 3). If the traversal reaches a leaf, the prediction value is

Algorithm 2: Synchronous Traversal (Device Side)

```
   Input: Worklist, Next_Worklist, Forest, Data, Output
1  for each thread t in GPU do in parallel
2      task ← Worklist.pop_element();
3      next_loc ← task.process_step(Forest, Data);
4      if next_loc is leaf then
5          atomicAdd(&Output[task.id],
               Forest.value(next_loc));
6      else
7          Next_Worklist.push({task.id, next_loc});
8      end
9  end
```

accumulated into the output array (line 5). Since multiple trees may contribute to the same query, we use `atomicAdd` to safely accumulate these partial results. Otherwise, the unfinished task is appended to the next-round worklist (line 7).

D. Level-based Forest Layout

Existing GPU engines, such as FIL [30], often use a DFS-like node ordering that groups nodes along root-to-leaf paths in memory. In contrast, DSTREE employs a level-synchronous workflow, processing a worklist of nodes at the same depth across multiple trees. This worklist-centric approach scatters same-depth nodes in a DFS-like ordering, leading to dispersed address access by warp threads and limiting memory coalescing and cache reuse. To align better with this access pattern, DSTREE adopts a level-based layout.

In this level-based layout, nodes are stored in *depth-major* order, with all nodes at depth (d) placed before those at depth (d+1). For instance, Figure 4a shows depth 0 nodes (0, 3), followed by depth 1 nodes (1, L1, L5, 4), then depth 2 nodes (2, L2, 5, L6), and finally depth 3 nodes (L3, L4, L7, L8). This organization confines each iteration's memory accesses to a compact range, enhancing spatial locality and enabling more coalesced memory transactions and cache reuse.

V. Optimizations for Efficient Worklist Processing

While the synchronous traversal framework reduces idle threads through dynamic task reassignment, it still reveals optimization opportunities: (1) lowering synchronization costs from the global barrier after worklist processing and (2) minimizing memory traffic from frequent shared worklist accesses. We propose a privatized synchronous traversal scheme and a dynamic reordering technique to improve worklist efficiency.

A. Privatized Synchronous Traversal

Figure 6 shows how privatized synchronous traversal works. Instead of setting a global barrier after every single traversal step, each thread will execute its task for S steps before synchronization. Figure 6 illustrates an example for setting S = 2. In the second step, thread 1 and thread 3 become idle because their tasks have already reached leaf nodes and require no further traversal. Therefore, privatized synchronous traversal essentially trades off thread utilization for reduced global synchronization overhead: as the step size S increases, the cost of global barriers and worklist accesses decreases, but thread underutilization increases. Specifically, when S = 1, the algorithm degenerates into the naive worklist-based approach described in Section IV-A. It is important to note that when $S = max\_depth$ (the maximum depth of the forest model), the algorithm degenerates to the traditional static task assignment strategy. Tuning S allows us to explore the tradeoff between synchronization overhead and thread utilization.

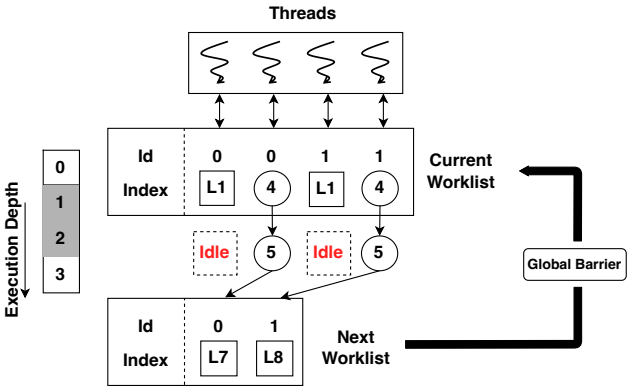


Fig. 6: Privatized Synchronous Traversal.

Algorithm 3: Privatized Synchronous Traversal (Device Side)

```
   Input: Worklist, Next_Worklist, Forest, Data, Output
1  for each thread t in GPU do in parallel
2      task ← Worklist.pop_element();
3      for i ← 1 to S do
4          next_loc ← task.process_step(Forest, Data);
5          if next_loc is leaf then
6              Output[task.id] += Forest.value(next_loc);
7              break;
8          end
9          task.loc ← next_loc;
10     end
11     if task does not reach leaf then
12         Next_Worklist.push({task.id, task.loc});
13     end
14 end
```

In this design, the host-side algorithm (Algorithm 1) remains unchanged. Algorithm 3 presents the privatized synchronous traversal kernel design on the device side. In contrast to Algorithm 2, which advances each task by only one traversal step per iteration, this scheme allows each GPU thread to process a task for up to S consecutive steps before synchronization.

B. Memory Access Optimizations For Worklist Execution

While privatized synchronous traversals mitigate thread underutilization, the problem of uncoalesced memory accesses

persists. As revealed in Observation 2, static task assignment causes queries in the same warp to diverge towards distant nodes in the forest, leading to uncoalesced memory access thus degrades effective bandwidth utilization.

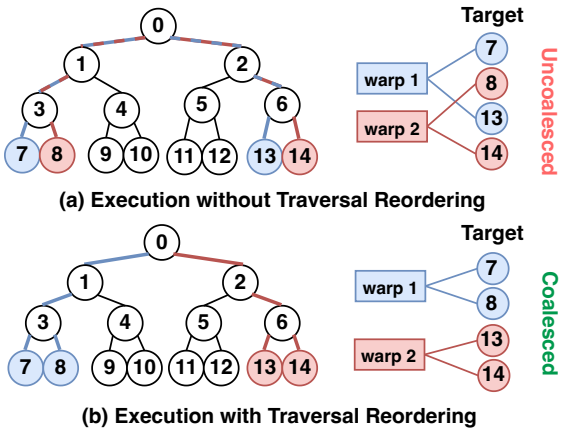


Fig. 7: Memory access patterns with and without traversal reordering. (a) Without traversal reordering, threads in the same warp reach distant leaves (7, 13) and (8, 14), resulting in uncoalesced memory accesses. (b) With traversal reordering, warp threads are assigned to nearby leaves, producing coalesced accesses. (Red and blue highlight execution path of different warps.)

We observe that, during actual inference, there exist opportunities where memory accesses from threads in the same warp can be coalesced if their traversal states are properly aligned. Figure 7 illustrates such a case. Without traversal reordering (Figure 7 (a)), warp 1 and warp 2 end up visiting distant leaves $7, 13$ and $8, 14$, which correspond to scattered memory locations. As a result, their accesses cannot be merged into a single transaction and become uncoalesced. In contrast, with traversal reordering (Figure 7 (b)), the same tasks are regrouped such that warp 1 targets $7, 8$ and warp 2 targets $13, 14$. Since these nodes are stored adjacently, memory requests fall into contiguous regions, allowing accesses to be coalesced.

To address this issue, our core idea is sorting. By reordering tasks so that memory accesses follow an ordered sequence, we can align thread requests to contiguous regions, thereby ensuring coalesced accesses. This transformation does not change the semantics of tree traversal, but rearranges the execution order to exploit memory locality.

A simple strategy is to reorder the entire task set after each inference step. We choose radix sort, as it outperforms other options in NVIDIA's CUB Library [34]. This reordering maximizes coalescing opportunities by ensuring warp threads access contiguous memory. However, full reordering is costly; Figure 8 shows that nearly 50% of the runtime is spent on sorting with a global sort at each step due to numerous tasks and frequent synchronization.

The cost of sorting primarily arises from two sources: (1) the overhead of the sorting kernel itself, and (2) the overhead

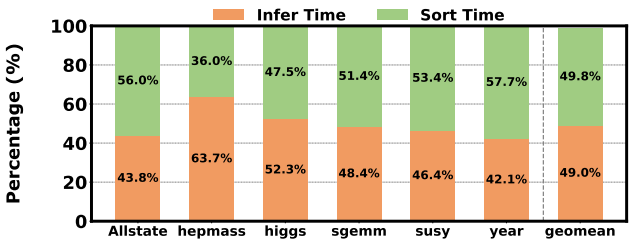


Fig. 8: Relative execution time breakdown between kernel computation and sorting at every traversal step.

incurred when inference execution is interrupted and replaced by barrier operations during sorting. Therefore, we design a series of mechanisms to reduce the overhead introduced by sorting operations.

Reorder Granularity. We consider two scopes: grid-level and block-level. Grid-level performs sorting over the entire worklist, which maximizes coalescing but also incurs the highest global-barrier and sorting costs. Block-level sorting conducts reordering within each thread block without global synchronization, thereby enhancing memory locality. To characterize the performance impact of this technique, we first introduce a tunable parameter, `block_sort_freq` to control the frequency of blockwise sorting: every `block_sort_freq` traversal steps, each block reorders its local worklist.

Reorder Scope. Full sorting processing ensures strict ordering of all tasks, but comes at a higher cost. Alternatively, partial sorting can be applied by exploiting parameters of radix sort, which tolerates some degree of disorder while still improving spatial locality. We introduce partial sorting with two parameters, `begin_bit` and `end_bit`, restricting the radix key to the bit range $[$`begin_bit`$,$ `end_bit`$)$, where only bits within this range are used as keys for sorting.

Reorder Frequency. In our privatized synchronous traversal, we must decide after each synchronization step whether to sort or skip sorting. Sorting enhances memory locality but adds overhead, while skipping reduces overhead at the risk of uncoalesced accesses. To address this, we introduce `global_sort_freq`, which controls grid-level reordering based on a specified number of synchronization steps.

Based on these observations, we balance memory coalescing benefits with reordering overhead using the four parameters introduced. Table I outlines their roles and trade-offs. These parameters are selected empirically. Figure 9 illustrates sorting's effect on performance speedup, with heatmaps showing results for the `higgs` dataset—greener indicates better performance, while red marks the optimal. Effective tuning enhances performance, validating our design's importance.

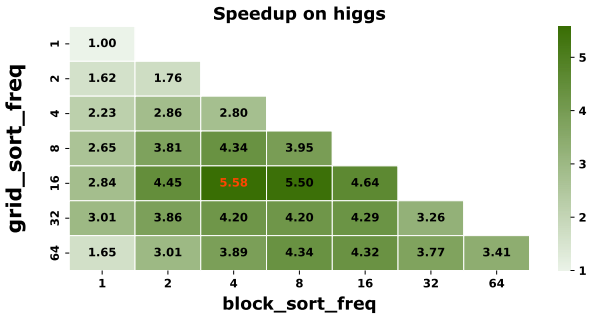
VI. Evaluation

A. Evaluation Methodology

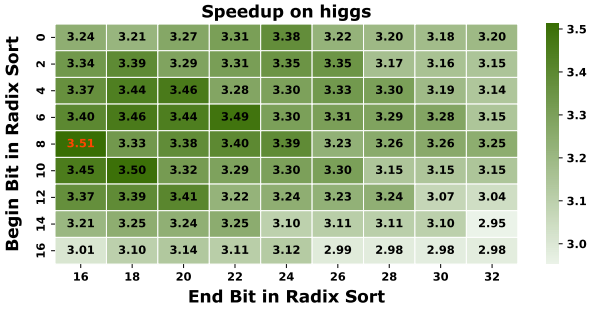
System Configurations. We conduct experiments on an NVIDIA RTX 4060 Ti GPU (8 GB DRAM, 34 SMs, 128

TABLE I: Trade-offs and control knobs across four customizable parameters.

Parameter	Function	Trade-off Controlled
global_sort_freq	Sets the total # steps per each grid-level sorting pass.	Global Ordering vs. Synchronization Overhead
block_sort_freq	Sets the total # steps per each block-level sorting pass.	Fine-grained Reordering vs. Reduced Local Cost
begin_bit	The number of leading bits skipped in radix sort.	Coarser Ordering Tolerance vs. Computational Overhead
end_bit	Defines the highest bit considered in radix sort.	Sorting Precision vs. Computational Overhead



(a) Effect of tuning global_sort_freq and block_sort_freq.



(b) Effect of tuning begin_bit and end_bit.

Fig. 9: Speedup is shown in each cell. The red cell highlights the best average performance.

KB L1 cache per SM, 32 MB L2 cache) [35] with a 13th Gen Intel Core i7-13700F host CPU. For comparison on CPU, we run the baseline implementations on a server-grade Intel Xeon Gold 6544Y processor (32 cores, 64 threads). We profile GPU kernels using NVIDIA Nsight Compute [36]. GPU kernel time is measured via CUDA events, averaged over 10 runs. Throughput is the number of queries processed per second. We use an NVIDIA RTX 3090 [37] for the portability study (24 GB DRAM, 82 SMs, 128 KB L1 cache per SM, 6 MB L2 cache) to evaluate the effectiveness of our proposal.

Benchmarks. We train the Random Forest for six datasets from `UCI` Dataset [38] and `Kaggle` [39] using the scikit-learn Python Library [32]. Table II shows the characteristics of the evaluated benchmarks.

We explore the effects of tree depth and ensemble size on classification tasks (`higgs`, `susy`, `hepmass`) and regression tasks (`Allstate`, `year`, `sgemm`) based on heatmap findings

TABLE II: Properties of the datasets used in experiments.

Name	# Features	# Classes	Task	Source
Allstate	33	N/A	Regression	Kaggle
hepmass	28	2	Classification	UCI
higgs	28	2	Classification	UCI
sgemm	14	N/A	Regression	UCI
susy	18	2	Classification	UCI
year	90	N/A	Regression	UCI

TABLE III: Evaluated schemes. We compare existing GPU inference engines with our proposed worklist-based variants. Abbreviations: global_sort_freq (*gs*), block_sort_freq (*bs*), begin_bit (*bb*), end_bit (*eb*)

Schemes	Descriptions
FIL [30]	State-of-the-art industry-quality decision forest inference engine on GPUs.
Tahoe [31]	GPU-based decision-forest inference engine using probabilistic layout optimization and adaptive inference strategies.
DSTREE	Our worklist-based design with tuned parameters. $gs \in \{1,2,4,8,16,32\}$; $bs \in \{1,2,4,8,16,32\}$ and $bs \leq gs$; $bb \in [1,16]$; $eb \in [16,32]$;

(Figure 10). Generally, deeper forests and larger ensembles improve accuracy, but overfitting occurs at high depths with few trees. For regression, increasing depth and size reduces the mean squared error (MSE) in `year` and `sgemm`, while `Allstate` reveals that larger ensembles stabilize performance, yet deeper models lead to higher MSE. We test benchmarks with tree counts ranging from 20 to 150 (in increments of 10) and maximum depths from 15 to 30 (in increments of 5).

Evaluated Schemes. Table III summarizes the evaluated schemes. For GPU baselines, we evaluate two prior inference engines: FIL [30], a state-of-the-art industry-quality library for decision forest inference, and Tahoe [31], a high-performance engine that accelerates decision-forest inference using probabilistic layout optimization and adaptive execution strategies. DSTREE is our worklist-based design with tuned parameters synchronization granularity, and reordering policies. Specifically, we explore a controlled parameter space to balance synchronization overhead and memory reordering cost. We vary the grid-level sorting frequency $gs \in \{1,2,4,8,16,32\}$ and the block-level sorting frequency $bs \in \{1,2,4,8,16,32\}$, where $bs \leq gs$. Additionally, we tune the radix-sort granularity using two parameters: $bb \in [1,16]$ and $eb \in [16,32]$, which define the `begin_bit` and `end_bit` of the sorting key. We per-

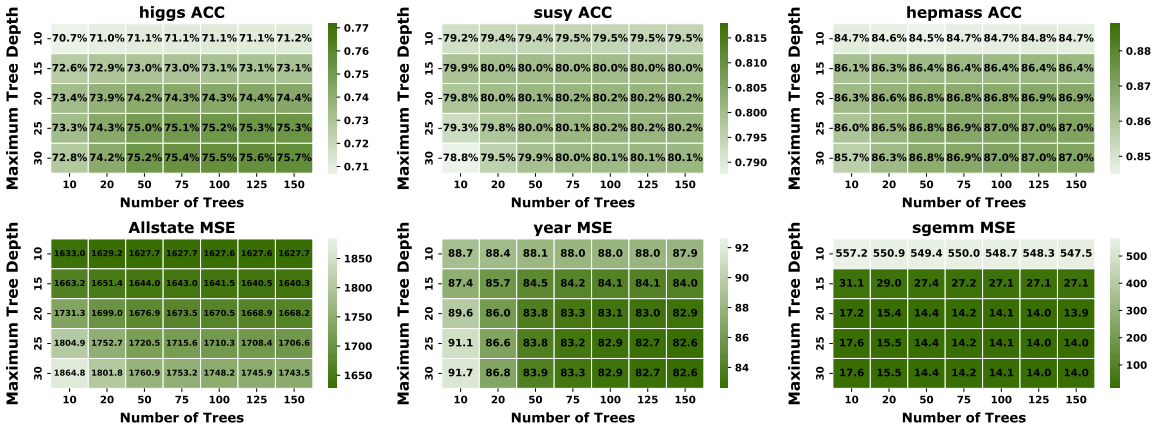


Fig. 10: Overview of the benchmark parameter choice with varying trees depths and number of trees. (Greener color indicates better model performance).

form a bounded search over the selected parameter space. The configuration that achieves the minimum runtime is reported as the final result of DSTREE.

B. Performance Results

Overall Performance Comparison with GPU baseline. Figure 11 compares the throughput of DSTREE and Tahoe [31] normalized to RAPIDS FIL [30]. We observe that DSTREE consistently delivers higher performance, and the performance gap widens as tree depth increases. At depth 15, the improvement is modest, with a geometric-mean speedup of only 1.07×. As the depth grows, the speedup becomes substantial, reaching 2.92× at depth 20, 3.57× at depth 25, and 3.94× at depth 30. Notably, DSTREE achieves a maximum speedup of 5.3× over FIL on `sgemm` when depth is increased from 20 to 30.

Unlike DSTREE, Tahoe shows only slight improvement over FIL for shallow depths and is unable to scale to deeper models. The challenge for Tahoe in executing models with deeper depth lies in its necessity to pad the trees into perfect binary trees, each containing $2^{depth} - 1$ nodes, running out of GPU memory. This shows that DSTREE greatly improves performance and scales efficiently.

Improvement Analysis. As shown in Figure 11, DSTREE yields little benefit for shallow trees (*depth*≈15) because thread utilization is already close to 100%. The worklist overhead may even negate gains. As depth increases, utilization drops (Figure 2), and DSTREE achieves larger speedups over static scheduling. On `year`, gains remain small due to its 90-dimensional features: improved model locality is offset by more random feature accesses.

Overall Performance Comparison with CPU baseline. Figure 12 compares the throughput of DSTREE normalized to scikit-learn [32] used as CPU baseline. As shown, DSTREE achieves an average of 50.0× speedup over scikit-learn, with up to nearly 74.3× on `sgemm` and `susy`. Overall, these results demonstrate that, compared to the CPU (Xeon Gold 6544Y), DSTREE can significantly improve performance, highlighting that GPUs are particularly well-suited for large-scale decision forest inference tasks.

TABLE IV: Breakdown Analysis.

Schemes	Solutions		
	Worklist Execution	Privatization	Dynamic Reordering
DSTREE-naive	✓		
DSTREE-O^1	✓	✓	
DSTREE-O^1O^2	✓	✓	✓

C. Effectiveness of Optimizations

To understand the impact of proposed optimizations for DSTREE, we evaluate them incrementally on the synchronous traversal with the naive worklist method proposed in Section IV-A. Figure 13 shows the throughput normalized to synchronous traversal with the naive method. Each optimization is adding on top of the former one (as listed in Table IV). We make the following observations: 1) DSTREE-O^1 with the privatized synchronous traversal (shown in Figure 13) yields an average throughput improvement of 3.1× compared to the naive worklist-based approach due to the global synchronization overhead reduction. 2) DSTREE-O^1O^2 further achieves an average throughput increases of 7.2× relative to the naive method by adding our dynamic traversal reordering strategy, demonstrating the effectiveness in memory access enhancement. Consequently, DSTREE achieves significantly higher effective memory bandwidth utilization, as shown in Figure 16, which directly contributes to the overall speedup.

D. Breakdown Analysis

To measure the overhead of our method, we conduct a breakdown analysis shown in Figure 14. The results indicate that the extra cost is less than 20% of the total computational expense, while decision forest traversal dominates, averaging 84.7% of overall processing time.

Utilization Improvement. We measure the utilization improvement of FIL and DSTREE with depth=30, as shown

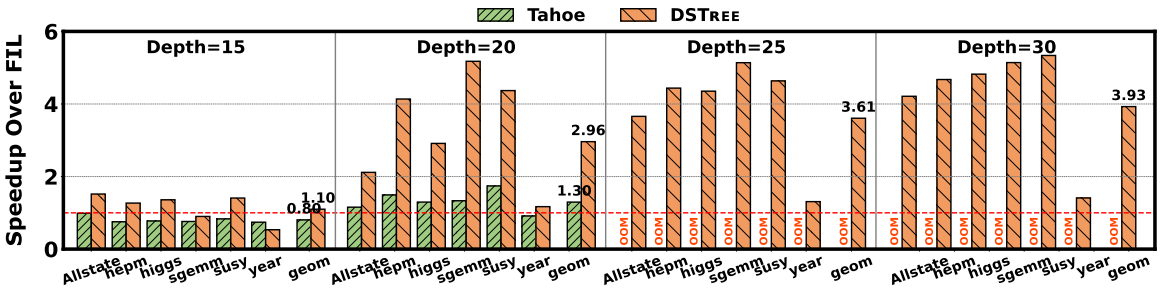


Fig. 11: Throughput results normalized to RAPIDS FIL [30]. Tahoe runs out of GPU memory (OOM) when dealing with deep tree forest traversals.

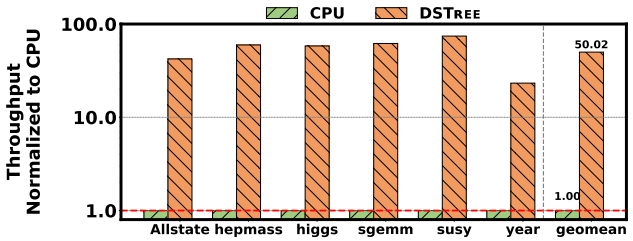


Fig. 12: Throughput results normalized to CPU baseline `scikit-learn` [32]. For each dataset, the reported value is the geometric mean speedup across all evaluated forest sizes (20–150 trees).

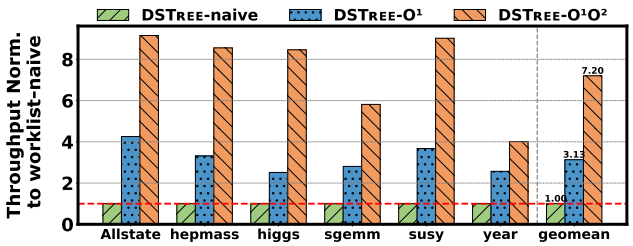


Fig. 13: Throughput normalized to DSTree-naive implementation.

in Figure 15. The privatized synchronous traversal method employed by DSTree improves thread utilization, increasing the average thread utilization from **61.16%** in FIL to **83.18%**. The thread utilization improvement in `sgemm` is marginal due to its shallower tree depth, which exhibits a balanced workload distribution pattern across threads. Thus, the static task assignment approach is already efficient. Overall, DSTree still enhances thread utilization and improves performance across diverse benchmarks.

Memory Bandwidth Utilization Improvement. We measure the memory bandwidth utilization improvement of FIL and DSTree. Figure 16 illustrates that FIL achieves an average utilization of only 30.9%, primarily due to its static scheduling, which results in highly irregular memory accesses. In contrast, DSTree effectively leverages the dynamic reordering technique to reduce uncoalesced memory access. This optimization substantially improves bandwidth efficiency, reaching an average utilization of 74.2%, with several datasets (e.g., `hepmass`, `susy`) sustaining over 85%. However, the `year` dataset shows only slight improvement due to its high input dimensionality, which creates significant data-access pressure. Dynamic task

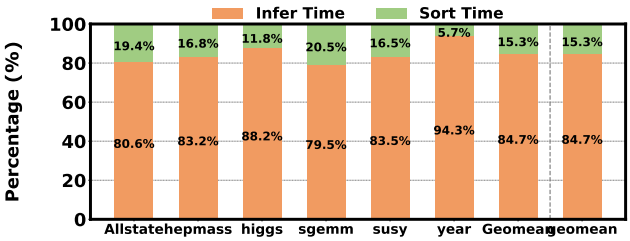


Fig. 14: Breakdown of DSTree execution time.

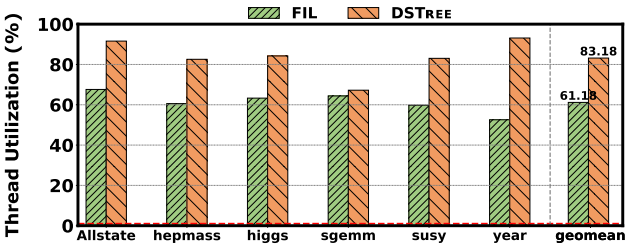


Fig. 15: Thread utilization improvement over FIL [30] at depth=30.

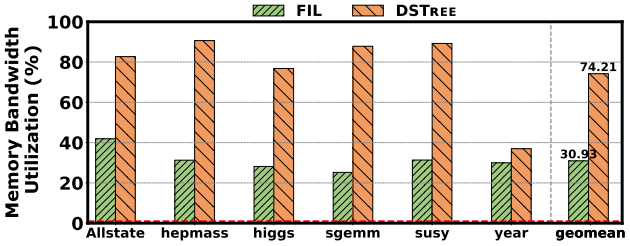


Fig. 16: GPU memory bandwidth improvement over FIL [30].

reordering enhances model data accesses but not query data access. Consequently, high-dimensional datasets see limited gains in memory bandwidth utilization.

E. Sensitivity Studies

We explore how performance sensitivity relates to two key factors: the number of trees and the synchronization interval (*steps*). This research aims to clarify how these parameters impact DSTREE and to identify scenarios where our optimizations are most beneficial.

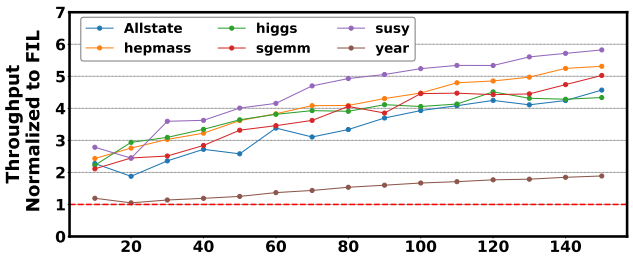


Fig. 17: Throughput results of DSTREE normalized to FIL [30] on different numbers of trees.

Sensitivity to Number of Trees. Figure 17 demonstrates the speedup of DSTREE compared to FIL across varying numbers of trees in the forest. As observed, the speedup ratio of DSTREE exhibits a monotonic increase as the number of trees grows. This phenomenon can be explained by the fundamental differences in memory access patterns between the two approaches. As tree depth increases, FIL [30] suffers from uncoalesced memory accesses, and the cost of strided accesses further increases with deeper forests (see Figure 3). In contrast, DSTREE leverages dynamic reordering to improve memory coalescing, still maintaining good performance for large forests.

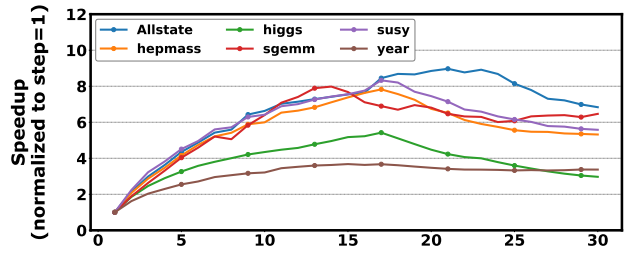


Fig. 18: Performance sensitivity of different steps (from 1 to 30). Results are geometric means across forest sizes (20–150 trees), normalized to step $s = 1$.

Sensitivity to Step Size. Figure 18 presents the sensitivity of throughput with respect to the synchronization interval s, normalized to the case $s = 1$. In this experiment, we focus on analyzing the benefit of the privatized synchronous traversal by varying s and examining its impact on performance. Across all datasets, performance steadily increases as s rises from 1, with fewer barriers reducing synchronization overhead. However, when s is too large, speedup declines due to static scheduling and resulting load imbalance issues. Optimal performance occurs at mid-range step sizes, typically ($1 \leq s \leq 30$), with most falling between 15 and 25. For instance, `Allstate` and `susy` achieve peak speedups (8×), while `higgs` and `year` show smaller gains due to higher memory intensity. These results highlight the need for balanced adaptive steps; excessive synchronization adds penalties, while insufficient synchronization hampers utilization.

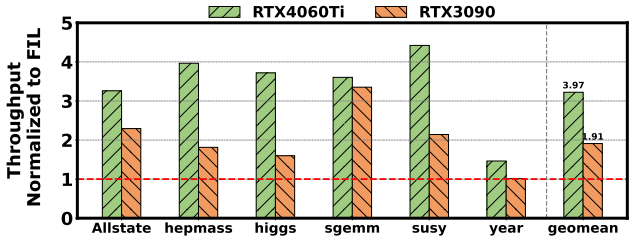


Fig. 19: Performance comparison of DSTREE on RTX 4060 Ti vs. RTX 3090.

F. Portability Studies

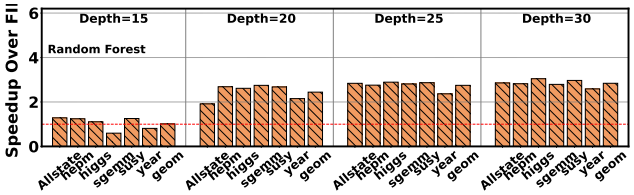
We further compare DSTREE and FIL on an RTX 3090 GPU. DSTREE outperforms FIL by 1.9× on an RTX 3090 GPU across all datasets, as shown in Figure 19. The speedup on `sgemm` is stable, but decreases on other datasets due to the RTX 3090's smaller L2 cache (6 MB) compared to the RTX 4060 Ti (32 MB). At an 100,000 batch size, only `sgemm` queries fit in the cache, increasing L2 cache miss rates for other datasets.

G. Discussion on Other Factors

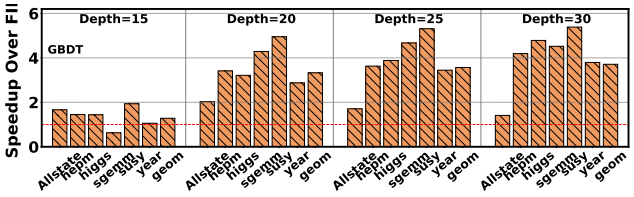
We conduct further experiments for a comprehensive evaluation of DSTREE. We analyze (1) larger memory GPUs to assess scaling and memory effects, (2) XGBoost-trained GBDT models to confirm generality across frameworks, and (3) the impact of batch size to understand performance sensitivity. These findings enhance insights into DSTREE's scalability and performance.

Experiments on GPUs with larger memory capacity. We conduct further experiments on an RTX 6000 Ada GPU, with results shown in Figure 20a. The results indicate an increasing speedup trend with tree depth: 1.01× for depth 15, 2.44× for depth 20, 2.75× for depth 25, and 2.84× for depth 30. Notably, the speedup for the `year` dataset is significantly greater on the RTX 6000 Ada compared to the RTX 4060 Ti (see Figure 19). This improvement is due to the RTX 6000 Ada's larger L2 cache (96 MB vs. 32 MB), which reduces performance degradation from random feature memory accesses.

Experiments using XGBoost's GBDT. We use XGBoost 1.7.4 to train GBDT models, tuning hyperparameters by varying maximum tree depth from 15 to 30 (in increments of 5) and tree counts from 20 to 150 (in increments of 10). Performance evaluation of the XGBoost models was



(a) Overall performance on the Random Forest benchmark



(b) Overall performance on the GBDT benchmark

Fig. 20: Performance analysis of DSTREE on different models on an RTX 6000 Ada GPU.

conducted on a RTX 6000 Ada GPU, with results shown in Fig 20b. DSTREE achieves significant speedup (geomean 3.71×) on GBDT models, demonstrating effective scheduling and memory access optimizations across ensemble types.

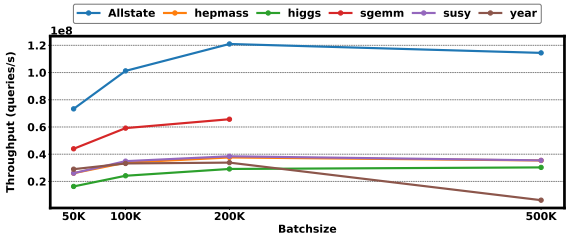


Fig. 21: Impact of Batch Size on Throughput of DSTREE

Experiments on Batchsizes. We examine how batch size affects the performance of DSTREE on the RTX 6000 Ada GPU with fixed parameters (100 trees and depth of 30). Figure 21 shows that as batch size increases from 50K to 200K, throughput improves consistently, indicating better GPU utilization from increased parallelism. However, beyond 200K, performance gains diminish, and for some datasets (e.g., year), performance slightly degrades. This decline is due to increased cache pressure, as larger batch sizes expand the working set, leading to more random memory accesses and reduced cache effectiveness. Ultimately, the penalties from random memory misses outweigh the benefits of improved memory coalescing. A moderate batch size offers the best balance between parallel efficiency and memory behavior.

VII. Related Work

Optimizations on Tree Traversal. A number of studies [40]–[49] have explored the acceleration of tree traversal operations. Among them, REGTT [41] employs worklists to carry traversal tasks across phases and uses a sort to reorder queries with similar truncation histories, improving thread utilization and memory coalescing. These studies primarily focus on optimizing recursive tree traversal methods, which are unsuitable for decision forest prediction scenarios.

Memory Optimizations on Decision Forest Inference. On CPUs, SIMD vectorization faces a key challenge in layout optimization. The goal is to design memory layouts that improve SIMD efficiency. Treebeard [50] proposes a tile-based memory layout. It partitions the decision trees in the forest into tiles and stores each tile in contiguous memory to support more efficient SIMD access. Van Lunteren [51] proposed an adaptive layout, which selects the appropriate memory layout at runtime according to the characteristics of the model. Chen et al. [52] and van Lunteren [51] both proposed optimizations that exploit the probability of traversing to a particular node in decision forests.

On GPUs, research has focused more on reducing uncoalesced memory access. Tahoe [31] introduces a probability-based forest layout optimization to improve memory coalescing based on the interleaved layout. Shah et al. [22] proposed a hierarchical layout optimization built on top of CSR layout.

Algorithm Optimizations on Decision Forest Inference. CPU work aims to boost SIMD efficiency. Treebeard [50] uses tile-based traversal for better inference, and Van Lunteren [51] applies an adaptive strategy to balance SIMD and multi-core parallelism in decision forest tasks.

On GPUs, Tahoe [31] eliminates reduction operations with three new algorithms. Shah et al. [22] optimize memory access by caching shallow layers in shared memory. Treebeard [50] employs speculative traversal, but this adds computation and memory access, making it unsuitable for GPUs. Traditional static traversal suffers from idle threads; in contrast, we improve thread utilization by dynamically reordering tasks.

VIII. Conclusion

We introduce DSTREE, a worklist-centric GPU framework for tree ensemble inference that restructures traversal into a level-synchronous execution model. DSTREE aligns execution granularity with memory organization and dynamically redistributes traversal tasks to improve thread utilization and memory locality, addressing the inefficiency of static, path-centric scheduling that amplifies warp divergence and uncoalesced memory accesses. Experimental results demonstrate that DSTREE substantially outperforms existing GPU inference engines, achieving up to **5.3**× speedup over RAPIDS FIL and **3.5**× over Tahoe. However, performance may degrade for high-dimensional inputs due to random feature access. Future work will investigate layout and cache-aware strategies to maintain high performance at large dimensionalities.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (No. 2024YFB4504200) and the Guangzhou-HKUST(GZ) Joint Funding Program (No.2025A03J3568).

REFERENCES

[1] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[2] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, 1995, pp. 278–282 vol.1.

[3] H. A. Salman, A. Kalakech, and A. Steiti, "Random forest algorithm overview," *Babylonian Journal of Machine Learning*, 2024.

[4] V. G. Costa and C. E. Pedreira, "Recent advances in decision trees: an updated survey," *Artificial Intelligence Review*, 2023.

[5] S. B. Kotsiantis, "Decision trees: a recent overview," *Artificial Intelligence Review*, 2013.

[6] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[7] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[8] A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble, "Overview of use of decision tree algorithms in machine learning," in *2011 IEEE control and system graduate research colloquium*, 2011.

[9] Q. Ren, H. Cheng, and H. Han, "Research on machine learning framework based on random forest algorithm," in *AIP conference proceedings*, 2017.

[10] Z. Yuan, X. Wang, Y. Nie, Y. Tao, Y. Li, Z. Shao, X. Liao, B. Li, and H. Jin, "Dynpipe: Towards dynamic end-to-end pipeline parallelism for interference-aware dnn training," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2025.

[11] H. Deng, "Interpreting tree ensembles with intrees," *International Journal of Data Science and Analytics*, vol. 7, no. 4, pp. 277–287, 2019.

[12] M. Z. Alam, M. S. Rahman, and M. S. Rahman, "A random forest based predictor for medical data classification using feature ranking," *Informatics in Medicine Unlocked*, vol. 15, p. 100180, 2019.

[13] M. Khalilia, S. Chakraborty, and M. Popescu, "Predicting disease risks from highly imbalanced data using random forest," *BMC medical informatics and decision making*, vol. 11, no. 1, p. 51, 2011.

[14] V. Podgorelec, P. Kokol, B. Stiglic, and I. Rozman, "Decision trees: an overview and their use in medicine," *Journal of medical systems*, 2002.

[15] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: massively parallel learning of tree ensembles with mapreduce," *Proceedings of the VLDB Endowment (PVLDB)*, 2009.

[16] H. Guan, S. Masood, M. Dwarampudi, V. Gunda, H. Min, L. Yu, S. Nag, and J. Zou, "A comparison of end-to-end decision forest inference pipelines," in *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*, 2023, pp. 200–215.

[17] A. Gershman, A. Meisels, K.-H. Lüke, L. Rokach, A. Schclar, and A. Sturm, "A decision tree based recommender system," in *10th International Conferenceon Innovative Internet Community Systems (I2CS)*, 2010.

[18] A. A.-U. Haque and H. Wang, "Rethinking conversational recommendations: Is decision tree all you need?" in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022.

[19] P. Thiengburanathum, S. Cang, and H. Yu, "A decision tree based recommendation system for tourists," in *2015 21st international conference on automation and computing (ICAC)*, 2015.

[20] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi, "A tensor compiler for unified machine learning prediction serving," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[21] A. Prasad, S. Rajendra, K. Rajan, R. Govindarajan, and U. Bondhugula, "Silvanforge: A schedule-guided retargetable compiler for decision tree inference," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 488–504. [Online]. Available: https://doi.org/10.1145/3694715.3695958

[22] M. Shah, R. Neff, H. Wu, M. Minutoli, A. Tumeo, and M. Becchi, "Accelerating random forest classification on gpu and fpga," in *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*, 2022.

[23] H. Zhang, S. Si, and C.-J. Hsieh, "Gpu-acceleration for large-scale tree boosting," *arXiv preprint arXiv:1706.08359*, 2017.

[24] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga?" in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012.

[25] F. Lettich, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Parallel traversal of large ensembles of decision trees," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[26] E. R. Gainza, C. Stewart, and A. O'Quinn, "Characterization of parallelism techniques for decision tree ensembles," in *2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2024.

[27] K. Jansson, H. Sundell, and H. Boström, "gpurf and gpuert: Efficient and scalable gpu algorithms for decision tree ensembles," in *2014 IEEE international parallel & distributed processing symposium workshops*. IEEE, 2014, pp. 1612–1621.

[28] Q. Wang, S. Cui, L. Zhou, Y. Dong, J. Bai, Y. S. Koh, and G. Russello, "Gtree: Gpu-friendly privacy-preserving decision tree training and inference," in *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2024, pp. 775–785.

[29] ——, "Xgt: Fast and secure decision tree training and inference on gpus," *IEEE Transactions on Dependable and Secure Computing*, 2025.

[30] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *arXiv preprint arXiv:2002.04803*, 2020.

[31] Z. Xie, W. Dong, J. Liu, H. Liu, and D. Li, "Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on gpu," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 426–440. [Online]. Available: https://doi.org/10.1145/3447786.3456251

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] J. Browne, D. Mhembere, T. M. Tomita, J. T. Vogelstein, and R. Burns, "Forest packing: Fast parallel, decision forests," in *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM, 2019, pp. 46–54.

[34] "Cooperative primitives for CUDA C++," https://nvidia.github.io/cccl/cub/, 2025.

[35] "NVIDIA ADA GPU ARCHITECTURE," https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf, 2023.

[36] NVIDIA Corporation, "NVIDIA Nsight Compute," https://developer.nvidia.com/tools-overview/nsight-compute/get-started, 2025.

[37] "NVIDIA AMPERE GA102 GPU ARCHITECTURE," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf, 2021.

[38] A. Asuncion, D. Newman *et al.*, "Uci machine learning repository," 2007.

[39] A. I. Company and J. Moser, "Allstate claim prediction challenge," https://kaggle.com/competitions/ClaimPredictionChallenge, 2011, kaggle.

[40] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for gpu execution of tree traversals," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2503210.2503223

[41] F. Zhang, P. Di, H. Zhou, X. Liao, and J. Xue, "Regtt: Accelerating tree traversals on gpus by exploiting regularities," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 562–571.

[42] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 15–22. [Online]. Available: https://doi.org/10.1145/1071866.1071869

[43] J. Liu, N. Hegde, and M. Kulkarni, "Hybrid cpu-gpu scheduling and execution of tree traversals," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA:

Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926261

[44] V. Singhal, L. Sakka, K. Sundararajah, R. Newton, and M. Kulkarni, "Orchard: Heterogeneous parallelism and fine-grained fusion for complex tree traversals," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, May 2024. [Online]. Available: https://doi.org/10.1145/3652605

[45] C. Koparkar, M. Rainey, M. Vollmer, M. Kulkarni, and R. R. Newton, "Efficient tree-traversals: reconciling parallelism and dense data representations," *Proceedings of the ACM on Programming Languages*, 2021.

[46] B. Ren, T. Mytkowicz, and G. Agrawal, "A portable optimization engine for accelerating irregular data-traversal applications on simd architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[47] M. Nam, J. Kim, and B. Nam, "Parallel tree traversal for nearest neighbor query on the gpu," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016.

[48] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "Simd parallelization of applications that traverse irregular data structures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[49] W. M. N. Zola, L. C. Bona, and F. Silva, "Fast gpu parallel n-body tree traversal with simulated wide-warp," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014.

[50] A. Prasad, S. Rajendra, K. Rajan, R. Govindarajan, and U. Bondhugula, "Treebeard: An optimizing compiler for decision tree based ml inference," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 494–511.

[51] J. Van Lunteren, "Accelerating decision-tree-based inference through adaptive parallelization," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2023, pp. 176–186.

[52] K.-H. Chen, C. Su, C. Hakert, S. Buschjäger, C.-L. Lee, J.-K. Lee, K. Morik, and J.-J. Chen, "Efficient realization of decision trees for real-time inference," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 6, Oct. 2022. [Online]. Available: https://doi.org/10.1145/3508019

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

IX. Overview of Contributions and Artifacts

A. Paper's Main Contributions

We propose DSTree, a worklist-centric GPU-based decision forest engine with a dynamic scheduling strategy that reassigns tasks at runtime to reduce idle threads and improve thread utilization. DSTree substantially outperforms existing GPU inference engines, achieving up to **5.3**× speedup over RAPIDS FIL and **3.5**× over Tahoe.

B. Computational Artifacts

This artifact includes the core components required to reproduce the main results of the paper. Specifically, it contains: 1) the setup of the execution environment and model building, 2) the analysis of thread utilization and memory access patterns, and 3) the performance evaluation of DSTree. These components reproduce data shown in Figure 2, Figure 3 and Figure 11.

The primary computational artifact is archived on Zenodo and made available on GitHub for any future updates or revisions, which are accessible through the following DOI link.

https://doi.org/10.5281/zenodo.18707788

X. Artifact Identification

Relation To Contributions

Since Artifact includes the source code and libraries, it supports all contributions described in the paper.

Expected Reproduction Time

- Preprocessing Stage (including model training, model format transformation, dataset format transformation, and source code compilation) takes about 19 hours, depending on the hardware configurations.
- Model Training and Data Preparation takes around 9 hours.
- Thread utilization and memory access pattern evaluations take around 10 minutes.
- Performance evaluation of DSTree on each model of GPUs takes around 47 hours.

Artifact Setup (incl. Inputs)

Hardware

Experiments require NVIDIA RTX 4060Ti GPU, RTX 3090 GPU, and RTX 6000 Ada GPUs systems. The detailed hardware configuration is shown as Table V.

TABLE V: Experimental Hardware Platforms.

	RTX 4060Ti	RTX 3090	RTX 6000 Ada
GPU Memory	8GB	24GB	48GB
GPU L2	32MB	6MB	96MB
CPU	i7-13700F	Xeon 4214R	2×6544Y
Cores/Threads	16/24	12/24	32/64
Max Freq	5.2GHz	2.4GHz	4.1GHz
CPU Cache (L2/L3)	24/30MB	12/16.5MB	64/90MB
Memory	32GB	128GB	512GB

Software

The experiments rely on the following software components.

- Ubuntu 22.04.5 LTS
- CMake 4.1.2
- Python 3.10
- Conda 25.7.0
- GCC 11.4
- CUDA 12.8

Datasets / Inputs

The datasets employ a suite of representative benchmarks across varying dimensions and sizes that are listed in Table II.

Specifically, the `hepmass`, `higgs`, `sgemm`, `susy`, and `year` datasets are publicly available from the UC Irvine Machine Learning Repository (https://archive.ics.uci.edu/), while the `Allstate` dataset can be obtained from the `Kaggle` platform (https://www.kaggle.com/competitions/ClaimPredictionChallenge/).

Artifact Execution

The artifact comprises three independent evaluation tasks for reproducing evaluation results:

1) Download the datasets and train forest models for DSTree.

2) Evaluation of thread utilization and memory access pattern (reproduces Figure 2 and Figure 3).

3) Performance of DSTree (reproduces Figure 11).

Expected Results

This artifact should show that our design achieves an average of 3.93× higher throughput on an RTX 4060Ti GPU, and 2.84× on an RTX 6000 Ada GPU, compared to state-of-the-art approaches FIL [30].

Artifact Analysis (incl. Outputs)

The outputs of the experiments are organized into three directories: `logs/`, `results/`, and `figures/`. The `logs/` directory stores the raw execution log files generated during both preprocessing and evaluation. The `results/` directory contains the processed outputs derived from these raw logs. The `figures/` directory stores the final visualizations and plots generated from the processed results.

Artifact Evaluation (AE)

Artifact Setup (incl. Inputs)

Before conducting the experiments, please ensure that CUDA 12.8 and Conda 25.7.0 are properly installed.

Artifact Execution

Evaluation 1: Prepare runtime environment, download the datasets, and train forest models for DSTREE

- **Create the Environment**: Set up the Conda environment.
- **Model Training and Data Preparation**: Train the models and transform both the trained models and datasets into the required internal formats by `preprocess.sh`.

The detailed instructions are as follows.

```
# Set up the preprocessing environment
$ bash scripts/setup_preprocessing.sh
# Set up the processing environment
$ bash scripts/setup_processing.sh
# Set up the cuml runtime environment
$ bash scripts/setup_motivation_cuml.sh
# Set up environment for preprocess.sh
$ conda activate preprocessing
# Model training, data transformation, project
    build, etc
$ bash scripts/preprocess.sh
# Compile the source code
$ conda activate processing
$ bash scripts/compile.sh
```

The script `setup_preprocessing.sh` creates the preprocessing environment, which includes XGBoost 1.7.4 and `scikit-learn` 1.7.1 for model training.

The script `setup_processing.sh` configures the processing environment, which includes RAPIDS 23.10 and the plotting library `matplotlib` 3.10.7 for evaluation and visualization.

The script `setup_motivation_cuml.sh` downloads the source code of RAPIDS 23.10. The motivation data reported in the paper is collected by instrumenting and inserting profiling code into the RAPIDS 23.10 implementation.

The script `compile.sh` compiles all the source include Tahoe [31], FIL [30] and DSTREE.

For detailed environment configurations, please refer to `preprocessing.yaml` and `processing.yaml` in the `conda_env/` directory.

Evaluation 2: Evaluation of thread utilization and memory access pattern

We provide comprehensive scripts for overall performance comparison.

The detailed instructions are as follows.

- **Create the Environment**: Set up the Conda environment and compile the source code by `setup_processing.sh`.
- **Execute Evaluation**: Execute the provided script to run experiments.
- **Plot Figures**: Execute the provided plot script. The final figures will be generated as `figures/fig_2.pdf` and `figures/fig_3.pdf`.

```
# activate conda environment
$ conda activate processing
# Run motivation experiment
$ bash run_motivation.sh
# plot figures
$ bash scripts/plot_fig2fig3.sh
```

Evaluation 3: Overall Performance Comparison

The detailed instructions are as follows.

- **Execute Evaluation**: Execute the provided script to run experiments.
- **Plot Figures**: Execute the provided plot script. The final figures will be generated as `figures/fig_11.pdf`.

```
# activate conda environment
$ conda activate processing
# Run full experiment
$ bash run.sh
# plot figures
$ bash scripts/plot_fig11.sh
```