Tahoe: Tree Structure-Aware High Performance Inference Engine for Decision Tree Ensemble on GPU

Zhen Xie
zxie10@ucmerced.edu
University of California, Merced

Wenqian Dong
wdong5@ucmerced.edu
University of California, Merced

Jiawen Liu
jliu265@ucmerced.edu
University of California, Merced

Hang Liu
hliu77@stevens.edu
Stevens Institute of Technology

Dong Li
dli35@ucmerced.edu
University of California, Merced

Abstract

Decision trees are widely used and often assembled as a forest to boost prediction accuracy. However, using decision trees for inference on GPU is challenging, because of irregular memory access patterns and imbalance workloads across threads. This paper proposes Tahoe, a tree structure-aware high performance inference engine for decision tree ensemble. Tahoe rearranges tree nodes to enable efficient and coalesced memory accesses; Tahoe also rearranges trees, such that trees with similar structures are grouped together in memory and assigned to threads in a balanced way. Besides memory access efficiency, we introduce a set of inference strategies, each of which uses shared memory differently and has different implications on reduction overhead. We introduce performance models to guide the selection of the inference strategies for arbitrary forests and data set. Tahoe consistently outperforms the state-of-the-art industry-quality library FIL by 3.82x, 2.59x, and 2.75x on three generations of NVIDIA GPUs (Kepler, Pascal, and Volta), respectively.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; *Machine learning*.

Keywords: Decision Tree Ensemble, Decision Tree Inference, Tree Structure, Performance Model.

ACM Reference Format:
Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. 2021. Tahoe: Tree Structure-Aware High Performance Inference Engine for Decision Tree Ensemble on GPU. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3447786.3456251

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456251

1 Introduction

Decision trees are among the most widely used machine learning models in practice [6, 20, 37, 41]. Decision trees have been widely used in many fields (e.g., advertising systems [12, 50] and medical diagnosis [5, 43]) and in enterprises. For example, Facebook uses high-throughput tree inference engines on GPU [33] to decide which notifications to send to billions of users; NVIDIA uses a high-throughput tree library (FIL) [10] on GPU to serve massive inference requests.

Decision trees are often assembled as a forest to boost prediction accuracy. With a decision tree ensemble, the predictions made by individual trees are combined together to make the final prediction. The prediction made by an individual tree is a path from the tree root to a tree leaf. The path is typically composed of an arrangement of choices. Each choice has the form "$x_j < b_i$", asking whether the attribute x_j is less than a threshold b_i. Hence, the path includes a series of IF-THEN rules, each of which is represented by a tree node.

The above characteristics of decision tree ensemble create major obstacles to implement high throughput inference (i.e., processing inference requests as many as possible within a given time). In particular, to enable high throughput inference for massive inference requests in common use cases, the user often uses GPU where each GPU thread makes prediction using one or more trees for one or more inference requests. Leveraging massive thread-level parallelism and high memory bandwidth, the user expects GPU to provide high throughput inference. However, different GPU threads traverse different trees, and can take different tree paths and access different attributes of input samples during the forest traverse, which causes irregular memory access patterns. Memory accesses from threads tend to be uncoalesced, which leads to poor performance, low hardware utilization and underutilized memory bandwidth. Using an industry-quality inference engine (i.e., FIL in NVIDIA RAPIDS suite [10]) to evaluate decision trees on GPU, we show that the ratio of requested data to total fetched data from global memory is only 27.2%, because of uncoalesced memory accesses.

Furthermore, there is a load imbalance problem across threads, which makes the above performance problem even

worse. In particular, during the construction of decision tree ensemble (i.e., tree training), attributes of training samples can be randomly selected [22] and some post-pruning techniques [19, 42] can be applied to the trees to enhance generality and prevent overfitting. As a result, trees in ensemble have different depths, and threads assigned with different trees to traverse have different workloads. Such workload imbalance across threads causes thread idling and low hardware utilization. Our preliminary work on FIL shows that execution time across threads has up to 10x difference.

Using GPU for decision tree inference has not been explored well, and the existing solution (i.e., RAPID FIL [10]) cannot fully address the above problems. To address the memory uncoalescing problem, FIL employs a storage format that stores nodes of different trees in an interleaved way, such that threads accessing nodes of different trees have a high chance of coalescing their memory accesses. This method, however, assumes that trees have similar structures and traversing different trees takes similar paths. These two assumptions do not often hold. We also notice that with FIL, the memory uncoalescing problem becomes more serious as trees are traversed near the leaves, because the chance that the trees are traversed via the same path towards the leafs becomes smaller, as the trees are traversed deeper.

The fundamental reason accounting for the ineffectiveness of the existing solution is the ignorance of *tree structure* in decision tree ensemble. The tree structure not only means tree topology (e.g., tree depth), but also means what are the common paths to traverse the tree. If two trees have similar structures, they have analogous topology *and* common paths during tree traversing. Leveraging the tree structure information, we can assign trees between threads in a balanced way and arrange tree nodes and trees in storage in a way that memory accesses to trees tend to be coalesced.

In particular, we rearrange tree nodes in memory using the information of the tree structures, such that those nodes from different trees that tend to be accessed at the same time from multiple threads will be rearranged in contiguous memory space. This method in nature enables similarity between tree structures of different trees, which leads to highly coalesced memory accesses. Furthermore, we rearrange trees in tree ensemble, such that trees with similarity structure are grouped together in memory and assigned to threads in a balanced way. However, determining the similarity between trees is challenging, because of high computation cost of pairwise comparison. We employ SimHash [8] and Locality Sensitive Hashing [21] to enable efficient comparison between different trees to determine their similarity.

Besides being aware of tree structure, making the best use of shared memory is also critical to improve inference performance. Either input samples or trees can be placed into shared memory (but not both, because of limited capacity of shared memory). Depending on tree structures, sample properties (e.g., sample size), and frequency of thread-blockwise reduction, different data placement strategies can lead to different inference performance. The traditional inference algorithm uses a strategy that places samples in shared memory, and cannot generally perform well for various tree ensembles. Thus we introduce three new data placement strategies. Given a tree ensemble and dataset, in order to decide which strategy performs best, we introduce performance models to predict performance and select optimal strategy with negligible cost. This approach, in combination with our adaptive forest format that consists of node rearrangements, tree rearrangements, and variable-length representation, leads to a tree structure-aware high performance inference engine, named *Tahoe*.

In general, Tahoe is input data-aware and architecture-aware due to the adaptive forest format and three inference strategies respectively. The performance optimization techniques employed by Tahoe have potential to be applied to other applications with irregular data structures (e.g., regular expression matching [35]) to improve performance on GPU.

We summarize the major contributions as follows.

- We introduce an inference engine, Tahoe, for decision tree ensemble on GPU; Tahoe is adaptive to various tree structures by re-arranging node and tree layout in memory to improve memory access efficiency and avoid load imbalance, and by using the optimal data placement strategy to make best use of shared memory and reduce parallel reduction overhead.
- We evaluate Tahoe with 15 common datasets on three generations of GPU based on Kepler, Pascal and Volta microarchitectures. Tahoe consistently outperforms the state-of-the-art industry-quality inference engine FIL. Compared with FIL, Tahoe leads to 5.31x, 3.67x and 4.05x speedup (up to 9.58x, 8.77x, and 10.14x) for high parallelism tasks and 2.34x, 1.52x and 1.45x speedup (up to 5.08x, 3.82x, and 3.17x) for low parallelism tasks on Kepler, Pascal and Volta GPUs, respectively.

2 Background

In this section, we introduce background information.

Decision tree and ensemble. A decision tree is a decision support system that uses a tree-like graph structure with various conditional branches. As a non-parametric supervised learning method, decision trees are often used to learn classification or regression function by piecewise constant function [24]. Tree#1 in Figure 1 is an example of a decision tree. This tree consists of a root node, a set of interior nodes, and leaf nodes that predict final outcomes. We study binary decision trees in this paper, because they are the most common ones. With a binary decision tree, each node has at most two children nodes. With a decision tree, each decision node (either the root or an interior node) contains an attribute index (e.g., $F1$ in the root of Tree#1) corresponding to one of the input variables in a sample for inference, a value (e.g., V_{11}

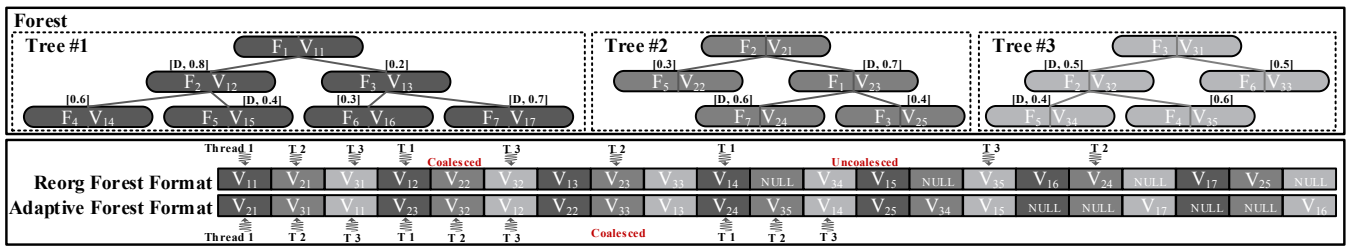

Figure 1. An example of decision tree ensemble and memory access sequences using the reorg forest format.

in the root of of Tree#1) to determine which path should be taken given an input, and a default path (D) to be taken when the attribute in the input does not have a value. After the training process, the probability that each edge of the tree is taken is calculated, and is used by Tahoe during inference. We name the probability that an edge is taken *edge probability* in the rest of the paper. The probability that a node in a tree is visited is the product of probabilities of edges from the root to the node. We name the probability that a node is visited *node probability* in the rest of the paper. The node possibility can be calculated based on edge probability.

Decision trees can form an ensemble, such as random forest [22] or Gradient Boost Decision Trees (GBDT) [15]. Within an ensemble, trees can differ from each other in terms of tree structure, tree depth and indexes of attributes in nodes. We use the terms "ensemble" and "forest" interchangeably in the following discussion.

Storage format. Trees are often stored in memory using a format called "reorg format" [10]. Given a tree ensemble to store, this format interleaves nodes of different trees. In particular, the root nodes of all trees are stored first, followed by left nodes at the second levels of all trees, and followed by right nodes at the second levels of all trees, and so on. The nodes for each tree are stored in breadth-first order.

Figure 1 gives an example of this format. This example is an ensemble of three trees. This example first stores the first level (root node) of three trees (particularly V_{11}, V_{21} and V_{31}), and then the left nodes at the second level of the three trees (particularly V_{12}, V_{22}, and V_{32}), and then the right nodes at the second level of the three trees (particularly V_{13}, V_{23}, and V_{33}), and then the nodes at the third level.

Tree inference algorithm on GPU. We discuss the state-of-the-art tree inference algorithm employed in FIL, an industry-quality tree inference engine used in NVIDIA RAPIDS [10]. With this inference algorithm (named *shared data*), each thread block accesses the whole tree ensemble; In each thread block, trees in the tree ensemble are evenly assigned to threads in a round-robin way; Threads in each thread block load as many samples as possible (i.e., a *batch* of samples) into shared memory for inference. Each batch of samples is just big enough to fill shared memory. Given a sample for inference, each thread in a thread block uses its assigned trees to make prediction, and then all threads in the thread block perform a thread block-wise reduction to compute the final prediction based on prediction results of individual trees. The final prediction is then written into global memory. Samples for inference are processed batch by batch, in order to leverage massive parallelism offered by GPU.

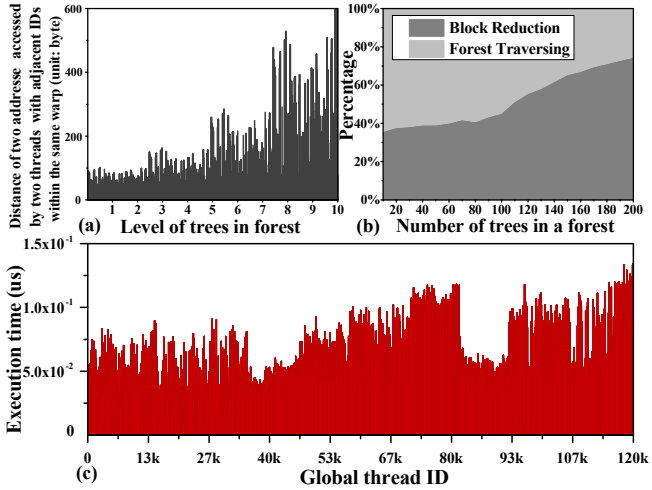

Figure 2. A motivating examples to reveal three performance problems: (a) uncoalesced memory accesses, (b) high reduction overhead, and (c) load imbalance across threads.

3 Motivation

Tree inference can lead to irregular memory accesses and fail to leverage full potential of high memory bandwidth of GPU. We study this problem using a random forest trained by XGBOOST [9] on a common dataset (Higgs [4]). We use 70% the dataset for training and 30% for inference. The forest has 120 trees and the maximum depth of each tree is 10. We use FIL as the inference engine that uses the shared data strategy as the inference algorithm. We identify three performance problems discussed as follows.

Uncoalesced memory accesses. Figure 2(a) shows the average distance of two addresses accessed by two threads with adjacent thread IDs within the same warp running the same instruction. Figure 2(a) shows the average distance at each level of trees. The figure shows that the average distance is small at the very beginning. This is due to the

effectiveness of the reorg format. This format interleaves nodes of different trees, such that when nodes are accessed by different threads taking the same branches (left or right), their memory accesses are coalesced. However, as the inference traverses into deeper levels of the trees, the average distance becomes larger. When the distance of two accesses is larger than the memory transaction size, memory uncoalescence happens. The distance becomes larger, because of two reasons: (1) As the trees are traversed deeper, the chance that different trees are traversed using different branches becomes higher; (2) trees have different typologies.

In this example, as the distance between two memory accesses is greater than 128 bytes (the size of a memory translation), the two memory requests are not coalesced into the same memory transaction. Using performance counters to measure efficiency of global memory accesses, we observe that at the tree levels of 7-10, the ratio of requested data to total data fetched from global memory is only 13.7%, because of uncoalesced memory accesses. In the rest of the paper, we refer *memory accesses efficiency* in terms of memory access coalesceness. We use the ratio of requested data to total data fetched from global memory as a metric to quantify memory access efficiency in global memory.

Load imbalance across threads. The tree inference in FIL assigns trees to threads in a thread block in a round-robin way without the awareness of tree balance. As a result, some threads are assigned with taller trees and perform more computation than other adjacent threads, causing load imbalance across threads. Figure 2(c) shows the execution time of threads in a thread block. The thread block performs inferences for 1000 samples. We use coefficient of variation (CV) to quantify the variance: CV=49.1%, indicating a large variance in execution time across threads.

High reduction overhead. The inference algorithm of shared data includes a block-wise reduction to compute the final prediction based on prediction results of individual threads. The reduction operation takes up to 53% of total inference time in our example (120 trees).

To further quantify the reduction overhead, we change the number of trees in the forest, re-train it, and measure the reduction overhead during inference. Figure 2(b) shows the reduction overhead when we change the number of trees from 10 to 200. The figure shows that the reduction takes 35%-72% of the total inference time; As the number of trees in a forest becomes larger, the overhead becomes larger.

The above three performance problems on GPU can be found in other applications as well, such as sparse matrix multiplication [23, 47–49], breadth-first search [26], scientific ML [13, 14], and cloud computing [46]. Addressing these problems is generally helpful to improve performance of those applications with irregular data structures.

4 Adaptive Forest Format

We introduce three techniques to address the problems of inefficient memory accesses and load imbalance.

4.1 Probability-based Node Rearrangement

The fundamental reason accounting for memory uncoalescing in the reorg format used in FIL is because threads working on different trees traverse different branches, while nodes at different branches of the different trees may not be laid out consecutively in memory space.

Take Figure 1 as an example again. In this example, we assume that the size of each tree node (including attribute index, threshold and default decision) is equal to one third of a memory transaction size. In other words, a memory transaction can contain data for up to three nodes. The nodes at the third level of the three trees in Figure 1 are stored in the following order using the reorg format, "V_{14}, NULL, V_{34}, V_{15}, NULL, V_{35}, V_{16}, V_{24}, NULL, V_{17}, V_{25}, NULL". When the three threads in the same thread block takes the left, right, and right branches to reach the third level respectively, "V_{14}, V_{24}, V_{35}" are accessed, which are not contiguous and does not fall into the same memory transaction.

To address the above problem, for each node with two children nodes, we ensure that the left child node always has higher possibility to be visited than the right child. Using this method, nodes that are at different trees (but at the same level of different trees) and have high probability to be visited are placed contiguously, so that they can be accessed by threads in a memory-coalesced way. The above method is implemented by swapping the two children nodes after tree training. In particular, based on the information of edge probability, if the left child has lower possibility to be visited than the right child, we swap the two children nodes; The descents of each child node go with the child node, as it is swapped. This method goes from top to bottom of the tree. We call this method the "probability-based node rearrangement".

We use the example in Figure 1 to further depict the above method. In Tree#2, the edge possibility of two children nodes of the root node at the second level is 0.3 and 0.7. The right child node (V_{23}) has higher possibility to be visited than the left child node (V_{22}). Hence, the two children nodes are swapped. The descents of V_{23} (i.e., V_{24} and V_{25}) go with V_{23}, as V_{23} is swapped. After swapping, the node V_{25} in Tree#2 and node V_{14} in Tree#1 are placed into contiguous memory space. When the two trees are traversed during inference, the branches that are taken and accessed are most likely the same, and the nodes accesses are likely coalesced.

In essence, the probability-based node rearrangement leverages data properties learned during the tree training to direct performance optimization during the tree inference. This is feasible, because a well-trained model is expected to see

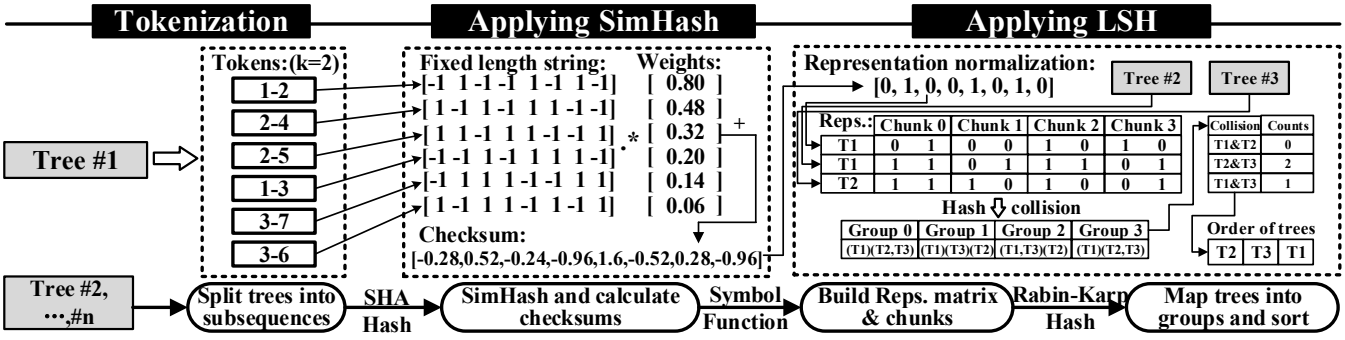

Figure 3. The workflow of similarity-based tree rearrangement. We use Tree#1 in Figure 1 as an example.

similarity in data properties between training and inference data sources in order to have high prediction accuracy.

4.2 Similarity-based Tree Rearrangement

We address the load imbalance problem based on the similarity of trees. We claim two trees are *similar*, if during inference, the two trees tend to be traversed using the similar paths and accessing similar attributes. We quantify the tree similarity online before any inference happens on a forest, which allows the usage of different tree rearrangements for a forests and enables the flexibility for incremental learning with the forest. The incremental learning can change the tree structures, and hence change the tree similarity accordingly.

To quantify similarity between trees and efficiently identify trees with high similarity in a given forest, we employ SimHash [8] and Locality Sensitive Hashing (LSH) [8]. The traditional method to quantify similarity between trees uses pairwise comparison [45], which leads to high computation complexity. Our evaluation shows that using pairwise comparison can take up to 19 minutes for a tree ensemble with 3000 trees. Such large overhead is too large for an online solution. Using SimHash and LSH, the complexity becomes much smaller (discussed later). SimHash and LSH have been very successful in the filed of information retrieval to find similarity among input information [38]. In our case, SimHash is used to transform trees to items such that we can compare the similarity of trees, and LSH is used to hash similar items into the same buckets, such that we can group multiple trees with high similarity into the same group. We describe our algorithm using SimHash and LSH as follows and call it "similarity-based tree rearrangement". Our algorithm includes three steps: tokenization, applying SimHash, and applying LSH. Figure 3 gives an example to depict the process of the rearrangement.

Tokenization. This step transforms the in-memory representation of each tree into a set of tokens, such that we can effectively use SimHash and LSH for data processing. In particular, given a tree, this step divides each path from the root to a leaf node into a set of tokens. Each token includes T_{nodes} nodes (T_{nodes} is a configurable parameter and we use $T_{node} = 2$ in Figure 3 as an example).

Applying SimHash. In this step, we first transform each token into a $1 \times L_{hash}$ string using a hashing algorithm (SHA1 [16]), where L_{hash} is the length of each token and equals to 8 in Figure 3 as an example. All strings of a tree have the same length, because of hashing. Each string is a vector of L_{hash} Boolean variables. We multiple each string with a weight which is the node probability of the last node in the token transformed to that string. Adding this weight is necessary to increase the effectiveness of LSH to capture similarity between trees. After that, all strings of a tree are added to generate a new string, which is the SimHash result of that tree. We name the SimHash result *checksum* in the rest of the discussion.

Applying LSH. In this step, each checksum is first normalized to a vector. This normalization regularizes each item of the checksum to either 0 (if the item value is less than 0) or 1 (if the item value is greater than or equal to 0). This normalization is necessary to efficiently apply LSH for the purpose of dimension reduction [25]. After normalization, each normalized checksum is evenly divided into M chunks ($M = 4$ in Figure 3 as an example). We apply a locality sensitive hashing (particularly the Rabin-Karp hashing) to each chunk. If two chunks from two normalized checksums has a hashing collision, then the two normalized checksums have similarity and their collision is counted. After applying LSH, trees are separated into buckets and trees in the same bucket has similarity. We count the number of collisions happened between each pair of trees in each bucket, and sort the number of collisions across buckets to decide which trees should be placed near each other. In particular, a tree A is placed near another tree B, when the number of collisions between A and B is the largest among the collisions between A and any other tree. After applying LSH, trees in the bucket are then assigned to threads in a round-robin way, such that loads are roughly balanced between threads.

Complexity analysis. The pairwise comparison leads to high computation complexity (particularly $O(2^{D_{tree}} * N^2_{trees})$,

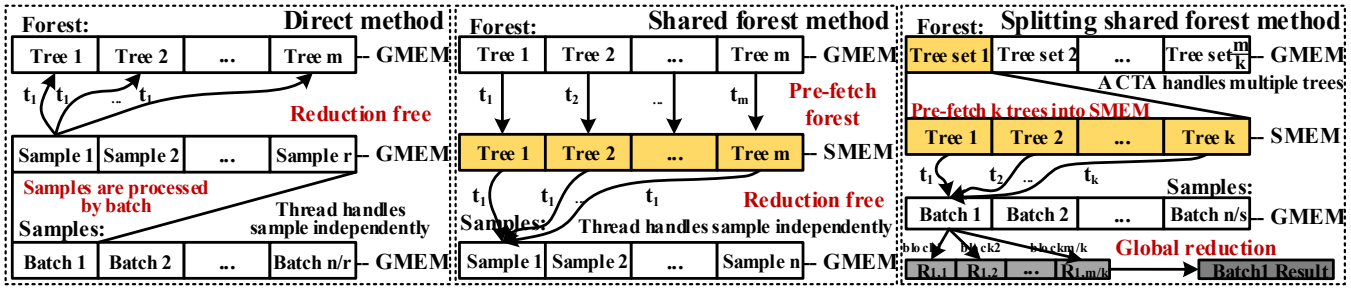

Figure 4. General descriptions for the three inference strategies. The usage of shared memory is highlighted in yellow

where D_{tree} is the average tree depth and N_{trees} is the number of trees) and high space complexity (particularly $O(N_{trees}^2)$). Using SimHash and LSH, the computation complexity becomes $O(D_{tree} * N_{trees}) + O(N_{trees})$, which is much smaller; The space complexity becomes $O(N_{tokens} * L_{hash}) + O(N_{trees} * M)$, where N_{tokens} is the number of tokens, L_{hash} is the length of each token, and M is the number of chunks. This space complexity is comparable or much smaller than that of the pairwise comparison, depending on the values of the parameters.

An example. In Figure 3, we have three trees from Figure 1. Each tree is tokenized into six tokens, each of which contains information for two nodes (T_{nodes} is set to 2). Each token is transformed into a 1×8 string (L_{hash} is set to 8), and each string is multiplied by a weight. After that, all weighted strings are accumulated together to build a 1×8 checksum as the result of applying SimHash. The checksum is then normalized to a vector of ones and zeros.

Given the three trees in the forest, we have three normalized checksums. Each checksum is divided into four 2×1 chunks. We apply LSH hashing to each chunk. If a chunk in a tree A has a collision with a chunk in another tree B, then the trees A and B are grouped into the same bucket and we count the collision in this bucket by one. After applying LSH, we get the number of collisions for each bucket. In our example, we have three buckets (T1,T2), (T2, T3), and (T1, T3), and the number of collisions for the three buckets are 0, 2, and 1 respectively. Based on the collected number of collisions, the order of trees laid in memory is T2, T3, and T1, because T2 and T3 have the largest number of collisions (or similarity), and T3 and T1 have the second largest similarity.

4.3 Adaptive Forest Format

Based on the above discussion, we introduce an adaptive forest format. The format stores the roots of all trees first (as the traditional reorg format). Different from the traditional format that place roots randomly, we decide which roots should be placed close to each other according to the results of similarity-based tree rearrangement. After the roots, we decide whether the order of left and right child nodes of each root should be switched, based on the probability-based node rearrangement. Besides the child nodes of roots, all other descendant nodes use the same method to be stored.

Besides the above node and tree rearrangements, we improve the representation of attribute index in the traditional storage format. The traditional storage format uses a fixed-length representation (usually four or eight bytes) to index an attribute in each node, no matter how many different attributes are there in the forest. Our storage format decides the length of the representation based on the number of different attributes in the forest. The length is just enough to index all attributes. Using this new representation instead of using a four-byte, fixed-length representation, we save storage space by up to 23.6% in our evaluated dataset. The new representation to index attributes in combination with tree and node rearrangement form an adaptive forest format.

5 Design of Inference Strategies

The adaptive forest format improves efficiency of global memory accesses and reduces load imbalance. In this section, we study how to address the problem of high reduction overhead shown in Figure 2(b). We also explore how to make best use of shared memory for high performance.

5.1 Inference Strategies

The existing shared data strategy loads data samples into shared memory and assigns a part of the forest to each thread. To make the final inference decision for a sample, all threads within a thread block must use a blockwise reduction operation. To avoid this reduction and explore performance potential of shared memory, we introduce several inference strategies as follows. These strategies use shared memory and reduction mechanism differently. Those three strategies are depicted in Figure 4.

Direct method. This strategy assigns the entire forest to each thread; Each thread loads the forest and samples from global memory. Shared memory is not used in this strategy.

This strategy completely removes blockwise reduction, because each thread has the entire forest to make the inference decision independently. However, with this strategy, all data accesses happen in global memory.

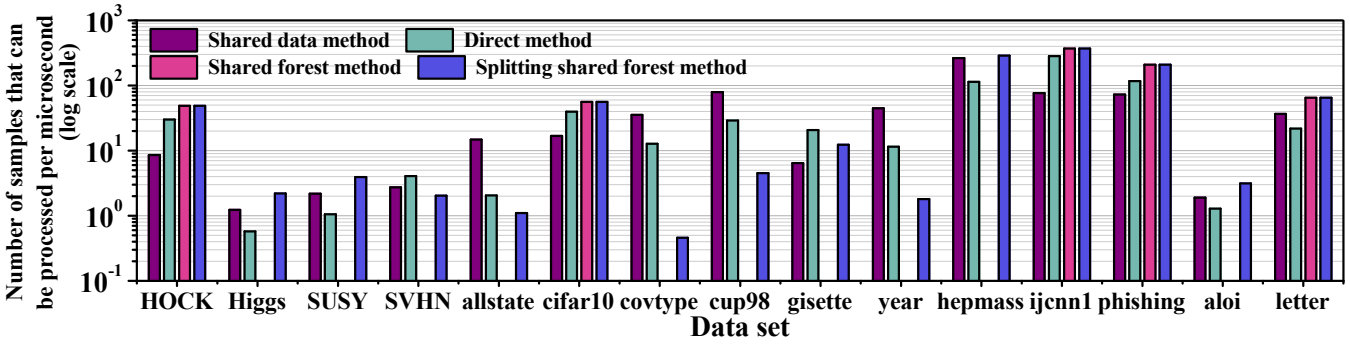

Figure 5. Performance comparison of the four inference strategies using 15 datasets on NVIDIA P100 GPU.

Shared forest. This strategy loads the entire forest into shared memory; Each thread reads samples from global memory, traverses the forest in shared memory and makes the final inference decision independently.

This strategy removes the blockwise reduction and does not have overhead of copying samples from global memory to shared memory; This strategy does not access the forest from the expensive global memory. However, the limitation of this strategy is that the forest must be small enough to be entirely loaded into shared memory. This strategy does not take advantage of shared memory to read samples.

Splitting shared forest. This is a variant of the shared forest strategy. The forest is split into P parts. Each part is a set of trees just big enough to be placed into shared memory. Each part is loaded from global memory to shared memory by a thread block (hence there are P thread blocks). Samples are loaded from global memory without being cached in shared memory. Each sample is assigned to P thread blocks. Each thread block processes sp samples in a batch and makes decisions for sp samples using its assigned trees in shared memory. There is a global reduction across P thread blocks to gather results from each thread block for sp samples to make final predictions. This splitting shared forest strategy addresses the limitation of shared memory capacity in the shared forest strategy, because of forest splitting.

5.2 Performance Comparison and Analysis

We study performance of the four inference strategies, i.e., the shared data method and three strategies proposed by us (the direct method, the shared forest method, and the splitting shared forest method). The four inference strategies use the adaptive forest format. For the shared forest method, if the forest cannot be entirely loaded into shared memory, the corresponding performance result is not shown. We use 15 datasets listed in Table 2. With each dataset, we use 70% of it for training and the remaining 30% for inference. We use XGBoost to generate and train forests. We use 100k as the sample batch size. We use NVIDIA Tesla P100 for evaluation. Figure 5 shows the results.

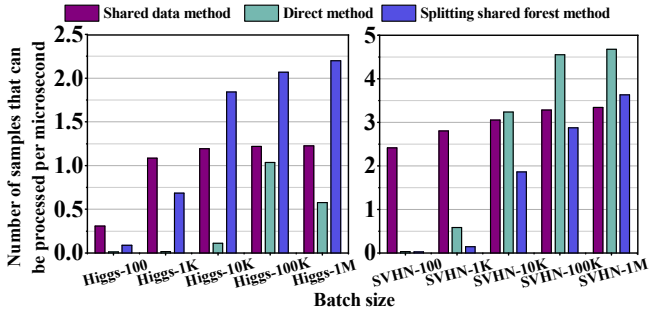

Figure 6. Performance comparison of the three inference strategies using five batch sizes (100, 1K, 10K, 100K and 1M) on two datasets (Higgs and SVHN).

The shared data method performs better than the other three strategies on four datasets (allstate, covtype, cup98, and year). The forests trained from these datasets are characterized with a relatively small number of trees: The number is less than 1K but these trees cannot be completely put into shared memory, and the number of attributes in each sample is relatively small. Given such characteristics, splitting shared forest performs worse, because it has to use one global reduction for each sample batch, which is more expensive than one blockwise reduction used in the shared data method. The direct method performs worse, because of frequent global memory accesses.

The direct method performs better than the other strategies on two datasets (SVHN and gisette). The forests trained from the two datasets consist of a set of tall trees. For such forests, the load imbalance across threads is serious (even with our tree rearrangement). As a result, the synchronization and reduction overhead take a larger portion of inference time. Removing them using the direct method is very beneficial.

The shared forest method can only be applied to five datasets (HOCK, cifar10, ijcnn1, phishing and letter), because of the limited shared memory capacity. The shared forest method performs better than the other strategies in the five datasets. This is not only because this method eliminates the

Table 1. Model notations and descriptions

Source	Symbol	Description	Source	Symbol	Description
Sample	S_{sample}	the size of a sample	Hardware parameters	BW_R_{SMEM}	Read bandwidth of shared memory
	N_{batch}	the number of samples in a batch		BW_W_{SMEM}	Write bandwidth of shared memory
Forest	D_{tree}	the depth of a tree		$BW\_R^{COA}_{GMEM}$	Read bandwidth of global memory for coalesced data
	N_{trees}	the number of trees in a forest		$BW\_R^{NCOA}_{GMEM}$	Read bandwidth of global memory for uncoalesced data
	S_{node}	the size of a decision node		$Num\_of\_threads$	the number of threads in a thread block
	S_{att}	the size of an attribute		$Num\_of\_thrd\_blocks$	the number of thread blocks
	N_{nodes}	the number of nodes in a tree		B_rate	Proportional parameter for block reduction
	S_{forest}	the size of a forest		G_rate	Proportional parameter for global reduction

blockwise reduction overhead, but also because the forest is reused more often than samples, and hence caching the forest instead of samples on shared memory is more beneficial.

The splitting shared forest method performs better than the other strategies in four datasets (Higgs, SUSY, hepmass, and aloi), because of the benefit of sharing forest instead of sharing samples in shared memory. Also, each tree in forests trained from the four datasets is relatively small and the amount of data that needs to be reduced by global reduction is small. This means the overhead of global reduction in the splitting shared forest is small.

We also change the sample batch size to study the impact of it on performance of the four inference strategies. We use five batch sizes, and use dataset Higgs and SVHN. The results are shown in Figure 6. We find that no inference strategies can consistently perform best for all batch sizes.

For example, for Higgs dataset, when the batch size is less than 10K, the shared data method performs best; Otherwise, the splitting shared forest performs best. The reason accounting for the above observation is as follows. When the batch size is small, the shared data method can effectively utilize shared memory to cache samples, which leads to high performance. As the batch size increases, the global reduction overhead for each sample in the splitting shared forest becomes smaller, because the overhead is amortized. As a result, the splitting shared forest method performs better than the shared data method.

Insight. No single strategies can perform best in all datasets with different batch sizes, datasets, and forests. Usage of shared memory and reduction overhead impact performance.

6 Tahoe: Tree Structure-Aware High Performance Inference Engine

Motivated by the insight in Section 5.2, we use performance modeling to decide which inference strategy should be used for best performance. Based on the performance modeling, we build a tree structure-aware high performance inference engine.

6.1 Performance Modeling for Tree Inference

Our performance modeling captures and quantifies performance critical operations, including shared memory and global memory accesses, blockwise reduction (using `cub::BlockReduce` [30]), and global reduction across thread blocks (using `cub::DeviceSegmentedReduce` [30]). The performance modeling does not consider cost of making decision at each tree node, because that cost is the same for all inference strategies. Table 1 lists model notations.

The execution time (inference time) T for processing *one* sample is modeled as follows:

$$T = T_{SMEM} + T_{GMEM} + T_{B\_REDU} + T_{G\_REDU} \quad (1)$$

where T_{SMEM} and T_{GMEM} are the execution time for accessing shared memory and global memory respectively; T_{B_REDU} and T_{G_REDU} are the execution time of blockwise and global reductions respectively.

We have three assumptions in the following models: (1) We assume that memory accesses to the *forest in global memory* are coalesced well, because of the adaptive forest format. This assumption is supported by our evaluation. Using micro-benchmarks to load forests based on memory access traces collected from inference, we find that the ratio of requested data to total fetched data from global memory to access the forests is 45.8%. Hence we assume that the bandwidth of memory accesses to the forest in global memory is equal to half of the bandwidth consumed by fully coalesced memory accesses. (2) We assume that memory accesses to samples to read attributes are fully random and no coalescence at all, because of the random nature of tree traverse; (3) We assume that there is little load imbalance between threads, because of similarity-based tree rearrangement. This assumption is supported by our evaluation: After the rearrangement, the coefficient of variation (CV) for execution time across threads is only 12.7%, which is four times smaller than that for the original forests. Besides the above assumptions, we ignore write accesses to global memory, because they are used to write prediction results and account for only a small portion (less than 1%) of total memory accesses.

T_{B_REDU} is modeled in Equation 2. T_{B_REDU} is proportional to the number of threads in a thread block. The ratio between T_{B_REDU} and $num\_of\_threads$ is a constant B_rate. B_rate is measured offline. T_{B_REDU} is divided by N_{batch} (where N_{batch} is the batch size), because we model the performance impact of the reduction on processing one sample.

$$T_{B\_REDU} = B\_rate * Num\_of\_threads / N_{batch} \quad (2)$$

T_{G_REDU} is modeled in Equation 3. T_{G_REDU} is proportional to the number of thread blocks. The ratio between T_{G_REDU} and $Num\_of\_thrd\_blocks$ is a constant G_rate. G_rate can be measured offline. T_{G_REDU} is divided by N_{batch}, because we

model the performance impact of reduction on processing one sample.

$$T_{G\_REDU} = G\_rate * Num\_of\_thrd\_blocks / N_{batch} \quad (3)$$

Based on the above discussion, we analyze performance of the four inference strategies as follows.

Shared data (using adaptive forest format). This strategy includes a blockwise reduction per sample, but does not have global reduction. We model the inference time for one sample as follows.

$$\begin{aligned} T_{SMEM} &= \frac{S_{sample}}{BW\_W_{SMEM}} + \frac{D_{tree} * N_{trees} * S_{att}}{BW\_R_{SMEM}} \\ T_{GMEM} &= \frac{S_{sample}}{BW\_R^{COA}_{GMEM}} + \frac{D_{tree} * N_{trees} * S_{node}}{(BW\_R^{COA}_{GMEM})/2} \end{aligned} \quad (4)$$

T_{SMEM} includes (1) the time of writing sample into shared memory (i.e., S_{sample}/BW_W_{SMEM}) after loading it from global memory, where S_{sample} and BW_W_{SMEM} are the sample size and write bandwidth of shared memory), and (2) the time of reading attributes of the sample from shared memory to meet the needs of traversing N_{trees} trees in the forest and the average depth of trees is D_{tree} (i.e., $(D_{tree} * N_{trees} * S_{att})/BW\_R_{SMEM}$, where BW_R_{SMEM} and S_{att} are the read bandwidth of shared memory and the size of an attribute).

T_{GMEM} includes (1) the time of loading the sample from global memory (i.e., $S_{sample}/BW\_R^{COA}_{GMEM}$, where $BW\_R^{COA}_{GMEM}$ is the read bandwidth of global memory under fully coalesced accesses), and (2) the time of traversing trees in global memory with improved memory coalescence using half of bandwidth (i.e., $(D_{tree} * N_{trees} * S_{node})/(BW\_R^{COA}_{GMEM})/2$, where S_{node} is the size of a decision node in trees).

With the shared data method, $T_{G\_REDU} = 0$ and there is one blockwise reduction per sample modeled in Equation 2.

Direct method. This strategy does not access shared memory ($T_{SMEM} = 0$) and is reduction free ($T_{B\_REDU} = 0$ and $T_{G\_REDU} = 0$). The inference time only includes T_{GMEM}.

$$T_{GMEM} = \frac{D_{tree} * N_{trees} * S_{node}}{(BW\_R^{COA}_{GMEM})/2} + \frac{D_{tree} * N_{trees} * S_{att}}{BW\_R^{NCOA}_{GMEM}} \quad (5)$$

T_{GMEM} includes (1) the time of loading the forest from global memory with improved memory coalescence (i.e., $(D_{tree} * N_{trees} * S_{node})/(BW\_R^{COA}_{GMEM})/2$) and (2) the time of reading attributes of the sample from global memory using uncoalesced accesses (i.e., $(D_{tree} * N_{trees} * S_{att})/BW\_R^{NCOA}_{GMEM}$).

Shared forest. This algorithm is reduction free ($T_{B\_REDU} = 0$ and $T_{G\_REDU} = 0$). We ignore the time of loading the forest from global memory to shared memory, because the forest is repeatedly used for inference after loaded and the loading time is easily amortized and ignorable. T_{SMEM} is the time of reading the forest in shared memory for inference; T_{GMEM} is the time of reading attributes in the sample in global memory using uncoalesced memory accesses.

$$\begin{aligned} T_{SMEM} &= \frac{D_{tree} * N_{trees} * S_{node}}{BW\_R_{SMEM}} \\ T_{GMEM} &= \frac{D_{tree} * N_{trees} * S_{att}}{BW\_R^{NCOA}_{GMEM}} \end{aligned} \quad (6)$$

Algorithm 1 Adaptive Inference Engine: Tahoe

1: Define: Input samples ($S_{samples}, N_{batches}$)
2: Define: Trained forest ($D_{tree}, N_{trees}, S_{node}, S_{att}, COA\_rate$)
3: Define: Hardware parameters (BR_{SMEM}, BW_{SMEM}, BR^{COA}_{GMEM}, BR^{NCOA}_{GMEM}, $Cost_{sync}$, B_rate, G_rate)

Hardware parameter detection (the offline part on CPU):

4: Measure hardware parameters by a microbenchmark

Optimization of forest format (the online part on CPU):

5: Fetch the tree ensemble and edge probability from GPU
6: Rearrange nodes, detect similarity, and convert the forest format
7: Send the forest format to GPU

Process of a batch for inference (the online part on GPU):

8: Predict performance of the shared data method using updated adaptive forest format by Equation 4
9: Predict performance of the direct method by Equation 5
10: **if** $sharedmemorysize > forestsize$ **then**
11: Predict performance of the shared forest method by Equation 6
12: **else**
13: Predict performance of the splitting shared forest method by Eq. 7
14: Set the maximum number of threads to hide latency, and set the number of blocks to maximize the occupancy of GPU register
15: Execute the method with the shortest predicted time
16: Count edge probabilities during inference

Splitting shared forest. This strategy performs similar to the shared forest, except including a global reduction. Such a reduction happens every N_batch samples, where N_batch is the number of samples in a batch. We divide T_{G_REDU} by N_batch to model the reduction time on a single sample.

This strategy needs to load the forest from global memory to shared memory every N_batch samples, because of the limited capacity of shared memory. The time of loading the forest is not ignorable, which is different from that in the shared forest. T_{GMEM} includes (1) the time of reading global memory to load the forest using coalesced memory accesses (i.e., $N_{nodes} * N_{trees} * S_{node}/BW\_R^{COA}_{GMEM}$) and (2) the time of reading attributes of the sample from global memory using uncoalesced memory accesses (i.e., $D_{tree} * N_{trees} * S_{att}/BW\_R^{NCOA}_{GMEM}$). T_{SMEM} includes (1) the time of writing the forest to shared memory after reading it from global memory (i.e., $N_{nodes} * N_{trees} * S_{node}/BW\_W_{SMEM}$) and the time of reading the forest in shared memory for inference (i.e., $D_{tree} * N_{trees} * S_{node}/BW\_R_{SMEM}$). We divide (1) in T_{GMEM} and (1) in T_{SMEM} by N_batch to model the performance impact of memory accesses on a single sample.

$$\begin{aligned} T_{SMEM} &= \frac{N_{nodes} * N_{trees} * S_{node}}{BW\_W_{SMEM} * N\_batch} + \frac{D_{tree} * N_{trees} * S_{node}}{BW\_R_{SMEM}} \\ T_{GMEM} &= \frac{N_{nodes} * N_{trees} * S_{node}}{BW\_R^{COA}_{GMEM} * N\_batch} + \frac{D_{tree} * N_{trees} * S_{att}}{B^{NCOA}_{GMEM}} \end{aligned} \quad (7)$$

6.2 Adaptive Inference Engine based on Performance Models

Algorithm 1 depicts the workflow of Tahoe. Tahoe consists of offline and online parts. In the offline part, Tahoe collects the values of those hardware parameters listed in Table 1 using

microbenchmarks. The offline part happens only once on a given platform. In the online part, Tahoe collects those sample and forest parameters listed in Table 1, rearranges tree nodes, detects the similarity of trees, and converts the forest. The above procedure happens as the system initialization before any inference happens. In the scenario of incremental learning, the above procedure is triggered whenever the forest is updated based on the learned new knowledge. We use CPU for the above procedure to avoid performance impact on GPU. At the inference time on GPU, Tahoe uses the performance models for once for each batch of samples to decide which inference strategy should be used for best performance. Using the above workflow, Tahoe becomes generally applicable to various hardware platforms, forests and samples. We quantify execution time of various components in Tahoe in Section 7.4.

7 Evaluation

7.1 Experimental Setup

Platform. We use three generations of NVIDIA GPUs, namely, Tesla K80 (Kepler), Tesla P100 (Pascal) and Tesla V100 (Volta) and an NVIDIA DGX-2 cluster. The machine with GPUs is equipped with two Xeon 8168 processors (each has 24 cores) running Linux 4.18.

Dataset. We use 15 datasets from UCI repository [4] and LIBSVM [7] for tree inference. The number of samples in those datasets ranges from thousands to millions; The number of attributes in those datasets ranges from tens to thousands. 70% of each dataset is used for training and 30% is used for inference. We build 15 forests, each of which is trained with one training set. The hyperparameters used for training those forests are based on [3, 31, 32, 39, 44] and Kaggle competition. Table 2 gives some details on those datasets and hyperparameters of the forests, including number of samples, number of attributes per sample, forest type, and maximum number of trees in a forest (N_{trees}) and maximum tree depth (D_{tree}) set by GBDT or random forest (RF).

Parameters in Tahoe. There are three parameters used in the similarity-based tree rearrangement: number of nodes per token (T_{nodes}) in the tokenization phase, length of each token (L_{hash}) in SimHash, and number of chunks (M) in LSH. We test various values of T_{nodes}, L_{hash} and M, and found that $T_{nodes} \in [4, 6]$, $L_{hash} \geq 128$ and $M \geq 64$ are usually sufficient to group these trees and give a correct order of trees based on their similarity. Hence, we choose $T_{nodes} = 4$, $L_{hash} = 128$, and $M = 64$ in our evaluation.

We analyze why choosing the three values make sense as follows. T_{nodes} has impacts on effectiveness and cost of SimHash. If T_{nodes} is too large, SimHash loses ordering accuracy; If T_{nodes} is too small, SimHash has longer execution time because too much tokens needs to be calculated. $T_{nodes} = 4$ leads to short execution time (about 0.226 ms on average) without losing accuracy. Using $L_{hash} = 128$ not only distinguishes different memory access sequences, but also makes hash-string aligned with the cache line size to improve performance. Using $M = 64$ is big enough to distinguish different buckets of trees.

Table 2. Details on datasets and decision tree ensembles. "RF" stands for random forest.

No.#	Dataset	#Samples	#Attributes	Forest type	N_{trees}	D_{tree}
1	HOCK	1993	4862	GBDT	8	8
2	Higgs	250000	28	RF	3000	8
3	SUSY	1000000	18	GBDT	2000	8
4	SVHN	1000000	3072	GBDT	218	15
5	allstate	588318	130	RF	800	5
6	cifar10	60000	3072	GBDT	10	8
7	covtype	581012	54	RF	500	3
8	cup98	17535	481	GBDT	150	8
9	gisette	13500	5000	GBDT	20	20
10	year	515345	90	RF	150	6
11	hepmass	10500000	28	GBDT	2000	10
12	ijcnn1	49990	22	RF	10	6
13	phishing	11055	68	RF	15	6
14	aloi	108000	128	RF	2000	6
15	letter	15000	16	RF	150	4

7.2 Overall Performance

Figure 7 shows throughput on all datasets. We use the performance of RAPIDS FIL as baseline for comparison. We show performance of using two batch sizes: 100K representing the use case of processing high-throughput tasks with high parallelism, and 100 representing the use case of processing low-latency tasks with low parallelism [10].

Figure 7 shows that Tahoe performs much better than FIL. For the high parallelism task, Tahoe introduces 5.31x, 3.67x and 4.05x speedup on average on K80, P100 and V100 GPUs respectively, compared to the FIL baseline; For the low parallelism task, Tahoe introduces 2.34x, 1.52x and 1.45x speedup on average on the three GPUs respectively. We have the following three observations: (1) Tahoe brings larger benefits to the high parallelism task than to the low parallelism task, because a high parallelism task allows more samples to be processed in parallel, which sufficiently leverages thread-level parallelism and amortizes the overhead of reduction. (2) K80 has higher speedup than other GPUs for low parallelism tasks, because K80 has smaller memory bandwidth and cache size, and K80 suffers more from memory uncoalesced problem, which provides more performance improvement opportunities. (3) V100 has higher speedup than P100, because V100 brings shorter inference time, which leads to more frequent occurrence of load imbalance (detailed data are shown in Table 3). The load imbalance brings performance improvement opportunities to Tahoe.

To quantify the contribution of three techniques to performance, i.e., (a) probability-based node rearrangement, (b) similarity-based tree rearrangement and (c) performance model-guided strategy selection), we apply the three techniques one after another. In particular, we apply (a), and then on top of (a), we apply (b), and then top of (a) and (b), we apply (c). Whenever we apply one technique, we measure the speedup using the performance of FIL as baseline, and calculate the speedup difference before and after using that

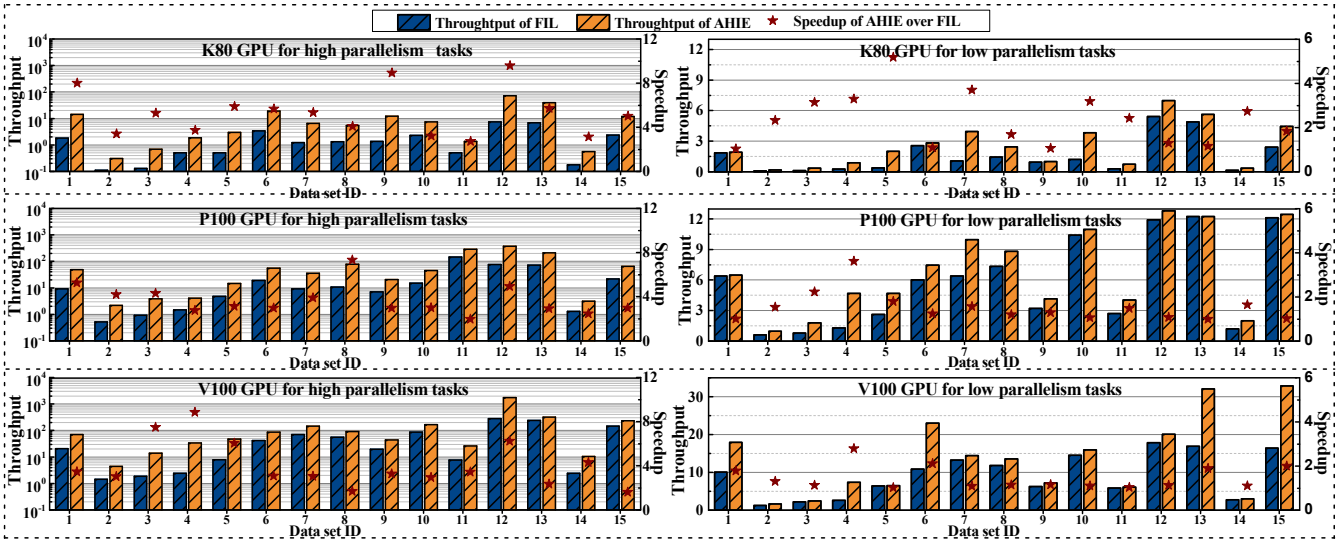

Figure 7. The performance of Tahoe and FIL on 15 datasets for tasks with high parallelism and low parallelism.

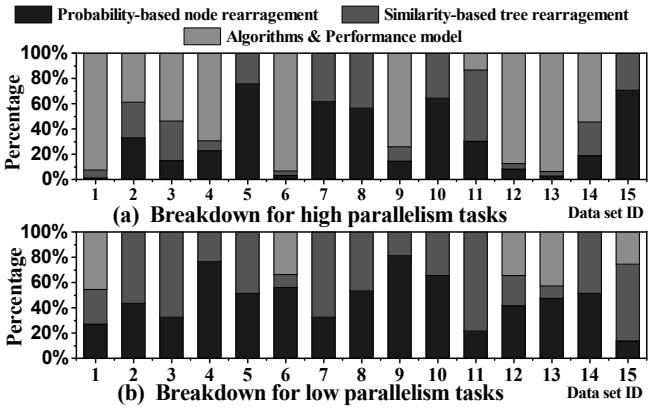

Figure 8. Quantifying the contributions of the three techniques to performance improvement.

technique; The speedup difference comes from the contribution of that technique. The results in Figure 8 are normalized by total speedup after applying all of the three techniques.

We have three observations. (1) The probability-based nodes rearrangement is very effective for the forests with shallow trees (e.g., those associated with the datasets 5, 7, 10 and 15), because shallow trees gives more opportunities to the node rearrangement technique to place nodes from different trees into the same memory transaction, enabling higher memory coalescing. (2) The similarity-based tree rearrangement is very effective for the forests with a large number of trees (e.g., those associated with the datasets 2, 3, 11 and 14), because when the number of trees is large, the load balance problem between threads is more serious. As a result, enhancing load balancing by the similarity-based tree rearrangement becomes very effective. (3) For the low parallelism tasks, the strategy selection tends to contribute less than the other two techniques, because the performance difference between the four inference strategies is small, making the strategy selection less effective.

Table 3. Quantifying load imbalance. "A.C.V." is the average coefficient of variation of execution time across threads.

GPUs	High parallelism tasks		Low parallelism tasks	
	A.C.V. of FIL	A.C.V. of Tahoe	A.C.V. of FIL	A.C.V. of Tahoe
K80	47.2%	13.1%	36.4%	10.8%
P100	51.3%	16.2%	42.9%	13.5%
V100	54.6%	15.9%	44.7%	12.5%

7.3 Performance Breakdown

Quantifying memory coalescence. We use NVProf [29] to quantify the load efficiency of shared memory and global memory when accessing forests. With Tahoe, the ratio of requested data from shared memory to total requested data is improved from 28.4% to 45.9% on K80, 28.7% to 48.3% on P100, and 29.7% to 50.6% on V100. The utilization of shared memory is significantly improved for high performance. Furthermore, with Tahoe, the global memory read throughput is improved from 62.4 GB/s to 174.7 GB/s on K80, 98.8 GB/s to 314.0 GB/s on P100, and 112.4 GB/s to 378.5 GB/s on V100. Such a large improvement in throughput comes from effectiveness of coalescing memory accesses. Based on the improvement of memory coalescence, there is large performance improvement for high and low parallelism tasks respectively (2.38x and 1.61x on K80, 1.85x and 1.22x on P100, and 1.98x and 1.21x on V100).

Quantifying load imbalance. We measure the average coefficient of variation (A.C.V) [1] of execution time across threads in a thread block in the 15 forests. Table 3 shows the results. Compared with FIL, the average coefficient of variation is reduced by 72.25% and 70.33% for K80, 68.42%

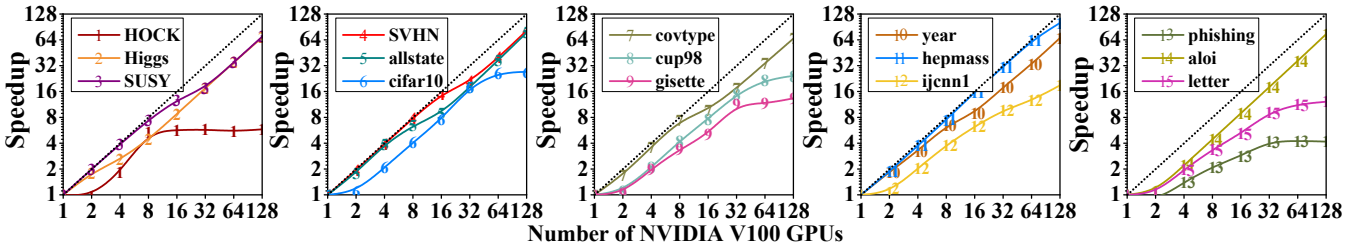

Figure 9. Strong scalability of the Tahoe frawework on 1-128 V100 GPUs using all 15 datasets.

and 68.53% for P100, and 70.88% and 70.04% for V100 for high and low parallelism tasks respectively. The similarity-based tree rearrangement leads to performance improvement of 2.03x and 1.51x for K80, 1.64x and 1.20x for P100, and 1.73x and 1.19x for V100 for high and low parallelism tasks. Forests trained from the datasets 7, 8 and 11 gain the largest benefit from load balancing, because there is large variance in tree depth across trees in each forest.

Quantifying effectiveness of removing blockwise reduction. Using performance-model guided strategy selection, Tahoe can reduce blockwise reduction overhead by using the three other inference strategies. Among 45 cases we evaluate with high parallelism tasks on the three GPUs, Tahoe removes blockwise reduction for 27 cases (Tahoe chooses the shared data method with blockwise reduction for other 18 cases for best performance); Among 45 cases we evaluate with low parallelism tasks, Tahoe removes reduction for 13 cases (Tahoe chooses the shared data method with blockwise reduction for other 32 cases for best performance). In general, the removal of blockwise reduction brings performance improvement of 2.94x and 1.16x for K80, 2.20x and 1.06x for P100, and 2.37x and 1.05x for V100 for high and low parallelism tasks respectively.

Quantifying the effectiveness of performance models. Tahoe uses the performance models to order performance of the four strategies. We evaluate if the performance models can correctly order the performance. Among 90 cases we evaluate using 15 datasets for high and low parallelism tasks on the three GPUs, 87 of them use the performance models to correctly order performance. Only three of them have slightly incorrect orders, but even with the orders, Tahoe still brings 4.73x, 2.71x and 2.26x performance improvement, which is close to the optimal (4.96x, 2.86x and 2.31x) manually selected. Such close performance is because the performance of those top-ranked strategies is close to each other in the three cases.

7.4 Overhead Analysis

We study **overhead of the CPU part of Tahoe**. This part includes (1) fetching the collected edge probability from GPU, (2) rearranging nodes of trees, (3) detecting similarity between trees, (4) converting the forest, and (5) copying the converted forest to GPU. The whole CPU part takes 28-57x as much time as one inference. The parts (1), (2), (3), (4) and (5) take 8-12x, 1-4x, 6-13x, 1-5x, and 11-15x as much time as one inference respectively. Compared with the pairwise comparison method, the part (2) reduces execution time by more than 37 times. The whole CPU part happens in parallel with inferences on GPU, hence its overhead can be easily hidden, because of short execution time. In addition, memory consumption of the adaptive forest is smaller than that of the original forest by 23.6%.

We study **runtime overhead of the GPU part of Tahoe**. The overhead includes running performance modeling for each batch of samples. Performance modeling consists of 8 addition, 26 multiplication, and 14 division for a total of 90 floating point operations, which takes only 0.92ns, 0.28ns, and 0.17ns on K80, P100 and V100 respectively. This overhead is an order of magnitude lower than the minimum inference time (25.42ns, 8.29ns and 5.63ns on K80, P100 and V100 respectively). Hence, the runtime overhead is very small.

7.5 Scalability Analysis

Strong scaling. We evenly partition each inference dataset into *NG* parts where *NG* is the number of GPUs. Each GPU gets one partition. We change *NG* for strong scaling tests, shown in Figure 9. Overall, Tahoe scales very well as the system scale increases. However, for some datasets (e.g., HOCK, gisette and phishing), the performance improvement is not scalable, as the system scale increases. This is because these datasets are relatively small. As the system scale increases, dataset per GPU becomes smaller and cannot offer enough thread-level parallelism to improve performance.

Weak scaling. We extend each inference dataset by *NG* times by randomly duplicating the dataset, and partition the duplicated dataset evenly between *NG* GPUs. We notice there is little performance variance (less than 5%) as the system scale increases (The results are not shown in the paper due to space limitation). Such good weak scaling results come from the fact there is almost no communication between GPUs.

8 Related Work

Decision tree ensemble. Decision tree ensemble can be divided into two main types: the independent and dependent ensembles. In the independent ensemble, each tree is built independently from others. In the dependent ensemble, the output of a tree affects the construction of the next tree. The

random forests proposed by Ho et al. [17] and Amit et al. [2] are typical independent ensembles. The Gradient boosting machine [15] (GBM) is a typical dependent ensemble. It trains each tree depending on trees that have already been trained. Each tree is built to maximize negative gradient of the loss function [28] and then pruned [11] to improve robustness. GBM usually has many shallow trees, as opposed to the random forest that has fewer but deeper trees.

Training decision tree ensembles. There are several algorithms to train tree ensembles. XGBoost [9] is a scalable boosting system that gains popularity in recent years. XGBoost applies a histogram and some refinements to GBM, aiming to reduce the size of training data to enable faster training. Lightgbm [19] is also a boosting tool. It uses a histogram-based tree splitting method to speed up the leaf split procedure and reduce memory consumption. CatBoost [34] is an efficient algorithm for vector representation of categorical data. Some existing work parallelizes decision tree training on multi-core CPU and many-core GPU [18, 27, 40].

Optimization of decision tree inference. FIL [10] is one of the few work focusing on improving throughput of decision tree inference on GPU. However, FIL has inefficient memory accesses, load imbalance and high reduction cost discussed in Section 3. Tahoe addresses these problems. Ren et al. [36] propose techniques to accelerate irregular data-traversal applications on SIMD architectures. Their work can be applied to improve inference performance. However, there are four fundamental differences between their work and Tahoe: (1) They only focus on data layout optimization, and their layout optimization does not consider tree similarity and does not *systematically* arrange tree nodes in all trees; (2) They heavily rely on the programmer to use an intermediate language to specify tree traversal, instead of automatically"learning" how to build layout as in Tahoe; (3) They do not consider the impact of sample/tree placement on memory and reduction overhead; and (4) They use a performance model to decide data layout, but the performance model only considers last level cache misses and is too heavyweight to be applied online.

9 Conclusions

Decision tree ensembles play an important role in many applications. However, how to use them efficiently for inference on GPU is challenging because of irregular memory access patterns and load imbalance across threads. This paper reveals that ignorance of tree structures is the fundamental reason accounting for the above performance problem. We introduce Tahoe, an inference engine on GPU that considers the common paths of tree traversal and the similarity of tree topologies to address the problem. Tahoe largely outperforms an industry-quality inference engine.

Acknowledgement. This work was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF1718194), and the Chameleon Cloud. We thank our shepherd, Paul Barham, for his constructive comments.

10 Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We conduct all experiments on a high-end server with 24 Intel Xeon E6-2760 v3 CPU cores running at 2.30GHz. The server is also equipped with three generations of NVIDIA GPUs, namely, Tesla K80 (Kepler), Tesla P100 (Pascal), and Tesla V100 (Volta). Some details of the experiments are listed below. 1) Speedup. We conduct experiments to test execution time by using our runtime algorithm, i.e., Tahoe and take the FIL as the baseline method. We use 15 input forests to evaluate the execution time. 2) We test the accuracy of performance model by running the 15 input forests on three GPUs. 3) We quantify the effectiveness of memory coalescence, load imbalance, and blockwise reduction. 4)We evaluate the performance of Tahoe framework in both strong and weak scaling.

10.1 ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are maintained in a public repository or are available under an OSI-approved license.

List of URLs and/or DOIs where software artifacts are available: https://github.com/zhen-xie/Tahoe.git

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: Some author-created data artifacts are maintained in a public repository or are available under an OSI-approved license.

List of URLs and/or DOIs where data artifacts are available: https://github.com/zhen-xie/Decision-tree-ensemble.git

10.2 BASELINE EXPERIMENTAL SETUP

Relevant hardware details: NVIDIA Tesla K80 (Kepler), NVIDIA Tesla P100 (Pascal), and NVIDIA Tesla V100 (Volta) GPU; Intel Xeon E6-2760 v3 CPU;

Operating systems and versions: Ubuntu 16.04 runing Linux kernel 4.4.0-206-generic;

Compilers and versions: g++ v5.4.0

Applications and versions: CUDA v11.0

Libraries and versions: RAPIDS FIL v0.18

Key algorithms: Locality-sensitive hashing;

Input datasets and versions: machine learning dataset (HOCK, Higgs, SUSY, allstate, cifar10, covtype, cup98, gisette, year, hepmass, ijcnn1, phishing, aloi, and letter)

References

[1] Hervé Abdi. 2010. Coefficient of variation. *Encyclopedia of research design* 1 (2010), 169–171.

[2] Yali Amit and Donald Geman. 1994. *Randomized Inquiries About Shape: An Application to Handwritten Digit Recognition.* Technical Report. CHICAGO UNIV IL DEPT OF STATISTICS.

[3] Kellie J Archer and Ryan V Kimes. 2008. Empirical characterization of random forest variable importance measures. *Computational Statistics & Data Analysis* 52, 4 (2008), 2249–2260.

[4] A. Asuncion and D.J. Newman. 2007. UCI Machine Learning Repository. http://www.ics.uci.edu/$\sim$mlearn/{MLR}epository.html

[5] Ahmad Taher Azar and Shereen M El-Metwally. 2013. Decision tree classifiers for automated medical diagnosis. *Neural Computing and Applications* 23, 7-8 (2013), 2387–2403.

[6] Christopher JC Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* 11, 23-581 (2010), 81.

[7] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 27.

[8] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 380–388.

[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.

[10] NVIDIA Corporation. 2020. RAPIDS | NVIDIA Developer. https://developer.nvidia.com/rapids. [Online; accessed 7-Jar-2020].

[11] Qun Dai. 2013. A competitive ensemble pruning approach based on cross-validation technique. *Knowledge-Based Systems* 37 (2013), 394–414.

[12] Dursun Delen, Cemil Kuzey, and Ali Uyar. 2013. Measuring firm performance using financial ratios: A decision tree approach. *Expert Systems with Applications* 40, 10 (2013), 3970–3983.

[13] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. 2019. Adaptive neural network-based approximation to accelerate eulerian fluid simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[14] Wenqian Dong, Zhen Xie, Gokcen Kestor, and Dong Li. 2020. Smart-PGSim: using neural network to accelerate AC-OPF power grid simulation. *arXiv preprint arXiv:2008.11827* (2020).

[15] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[16] Christian Henke, Carsten Schmoll, and Tanja Zseby. 2008. Empirical evaluation of hash functions for multipoint measurements. *ACM SIGCOMM Computer Communication Review* 38, 3 (2008), 39–50.

[17] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.

[18] Karl Jansson, Håkan Sundell, and Henrik Boström. 2014. gpuRF and gpuERT: efficient and scalable GPU algorithms for decision tree ensembles. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 1612–1621.

[19] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*. 3146–3154.

[20] Sotiris B Kotsiantis. 2013. Decision trees: a recent overview. *Artificial Intelligence Review* 39, 4 (2013), 261–283.

[21] Brian Kulis and Kristen Grauman. 2009. Kernelized locality-sensitive hashing for scalable image search.. In *ICCV*, Vol. 9. 2130–2137.

[22] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[23] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[24] Wei-Yin Loh. 2014. Classification and regression tree methods. *Wiley StatsRef: Statistics Reference Online* (2014).

[25] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 950–961.

[26] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 201–213.

[27] Aziz Nasridinov, Yangsun Lee, and Young-Ho Park. 2014. Decision tree construction on GPU: ubiquitous parallel computing approach. *Computing* 96, 5 (2014), 403–413.

[28] Alexey Natekin and Alois Knoll. 2013. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics* 7 (2013), 21.

[29] NVIDIA. [n.d.]. Profiler User's Guide, v10.2.89, 2020.

[30] NVlabs. 2018. CUB: a flexible library of cooperative threadblock primitives and other utilities for CUDA kernel programming. https://github.com/NVlabs/cub.

[31] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. 2012. How many trees in a random forest?. In *International workshop on machine learning and data mining in pattern recognition*. Springer, 154–168.

[32] Mahesh Pal. 2005. Random forest classifier for remote sensing classification. *International Journal of Remote Sensing* 26, 1 (2005), 217–222.

[33] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886* (2018).

[34] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. In *Advances in Neural Information Processing Systems*. 6638–6648.

[35] Bin Ren, Gagan Agrawal, James R Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10.

[36] Bin Ren, Todd Mytkowicz, and Gagan Agrawal. 2014. A portable optimization engine for accelerating irregular data-traversal applications on SIMD architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 2 (2014), 1–31.

[37] Shaoqing Ren, Xudong Cao, Yichen Wei, and Jian Sun. 2015. Global refinement of random forest. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 723–730.

[38] Ruslan Salakhutdinov and Geoffrey Hinton. 2009. Semantic hashing. *International Journal of Approximate Reasoning* 50, 7 (2009), 969–978.

[39] Robert E Schapire. 2003. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*. Springer, 149–171.

[40] Toby Sharp. 2008. Implementing decision trees and forests on a GPU. In *European conference on computer vision*. Springer, 595–608.

[41] Si Si, Huan Zhang, S Sathiya Keerthi, Dhruv Mahajan, Inderjit S Dhillon, and Cho-Jui Hsieh. 2017. Gradient boosted decision trees for high dimensional sparse output. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 3182–3190.

[42] Yan-Yan Song and LU Ying. 2015. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry* 27, 2 (2015), 130.

[43] Jyoti Soni, Ujma Ansari, Dipesh Sharma, and Sunita Soni. 2011. Predictive data mining for medical diagnosis: An overview of heart disease

prediction. *International Journal of Computer Applications* 17, 8 (2011), 43–48.

[44] Souhaib Ben Taieb and Rob J Hyndman. 2014. A gradient boosting approach to the Kaggle load forecasting competition. *International journal of forecasting* 30, 2 (2014), 382–394.

[45] Ferhan Ture, Tamer Elsayed, and Jimmy Lin. 2011. No free lunch: brute force vs. locality-sensitive hashing for cross-lingual pairwise similarity. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 943–952.

[46] Zhen Xie, Zheng Cao, Zhan Wang, Dawei Zang, En Shao, and Ninghui Sun. 2016. Modeling traffic of big data platform for large scale datacenter networks. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 224–231.

[47] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*. 94–105.

[48] Xie Zhen, Tan Guangming, and Sun Ninghui. 2021. Research on Optimal Performance of Sparse Matrix-Vector Multiplication and Convoulution Using the Probability-Process-Ram Model. *Journal of Computer Research and Development* 58, 3 (2021), 445.

[49] Xie Zhen, Tan Guangming, Liu Weifeng, and Sun Ninghui. 2020. PRF: a process-RAM-feedback performance model to reveal bottlenecks and propose optimizations. *HIGH TECHNOLOGY LETTERS* 3 (2020), 285–298.

[50] Ke Zhou, Shuang-Hong Yang, and Hongyuan Zha. 2011. Functional matrix factorizations for cold-start recommendation. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 315–324.