2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)

Treebeard: An Optimizing Compiler for Decision Tree Based ML Inference

Ashwin Prasad
Indian Institute of Science
Bangalore, India
and
National Instruments
Bangalore, India
ashwinprasad@iisc.ac.in

Sampath Rajendra, Kaushik Rajan
Microsoft Research
Bangalore, India
srajendra@microsoft.com
krajan@microsoft.com

R Govindarajan, Uday Bondhugula
Indian Institute of Science
Bangalore, India
govind@iisc.ac.in
udayb@iisc.ac.in

***Abstract*—Decision tree ensembles are among the most commonly used machine learning models. These models are used in a wide range of applications and are deployed at scale. Decision tree ensemble inference is usually performed with libraries such as XGBoost, LightGBM, and Sklearn. These libraries incorporate a fixed set of optimizations for the hardware targets they support. However, maintaining these optimizations is prohibitively expensive with the evolution of hardware. Further, they do not specialize the inference code to the model being used, leaving significant performance on the table.**

This paper presents TREEBEARD, an optimizing compiler that progressively lowers the inference computation to optimized CPU code through multiple intermediate abstractions. By applying model-specific optimizations at the higher levels, tree walk optimizations at the middle level, and machine-specific optimizations lower down, TREEBEARD can specialize inference code for each model on each supported CPU target. TREEBEARD combines several novel optimizations at various abstraction levels to mitigate architectural bottlenecks and enable SIMD vectorization of tree walks.

We implement TREEBEARD using the MLIR compiler infrastructure and demonstrate its utility by evaluating it on a diverse set of benchmarks. TREEBEARD is significantly faster than state-of-the-art systems, XGBoost, Treelite and Hummingbird, by 2.6×, 4.7× and 5.4× respectively in a single-core execution setting, and by 2.3×, 2.7× and 14× respectively in multi-core settings.

***Keywords*-Optimizing Compiler; Decision Tree Ensemble; Decision Tree Inference; Vectorization ; Machine Learning**

I. INTRODUCTION

Decision tree ensembles are among the most popular classes of machine learning (ML) models [1], [2]. The Kaggle state of ML survey 2021 [1] shows that more than 70% of data scientists use decision tree ensembles while less than 40% use neural networks. Such ensembles are generated by ML techniques like gradient boosting and random forests, reported to be the top two algorithms used in production pipelines at a large scale web company [2]. Such models also have many diverse use cases [3]. Not only are they used in computer science applications like search [4], recommendation and notification systems [5], but also in financial [6] and medical [7], [8] applications. Gradient boosted models (GBM) are even used in the CERN large hadron collider to classify particles [9].

A recent survey [10] that breaks down the cost of ownership of a data science solution among different stages indicates that inference costs have now become the dominant (in the range 45%-65%) component [11]. This motivates a careful study of the efficiency of inference with decision tree ensembles. Decision tree ensemble inference involves a simultaneous traversal of multiple decision trees. Given an input row x, a path from the root to leaf is traversed for each tree and the values at the leaves are aggregated together to produce a prediction. At each internal node an input feature x_i is compared with a threshold value v for that node to determine whether the walk moves left or right[1].

In this paper, we focus on optimizing decision tree inference on CPUs. CPUs are widely used for inference, both in the enterprise and in single machine use cases, because of their widespread availability and low cost [5], [12], [13]. Enterprise scale systems at several large companies frequently run decision tree inference on CPUs [14], [15]. Data science practitioners also predominantly run inference on commodity machines. Our analysis of solutions on Kaggle that use decision tree based models, including several competition winners [16], [17], reveals that CPUs are exclusively used to perform inference.

However, optimizing tree walks on a CPU is challenging because they have irregular access patterns and are sensitive to various architectural parameters. A profile of a basic traversal shows that a naïvely implemented tree walk has poor spatial and temporal locality, and therefore poor cache performance [18], [19]. Frequent branching and true dependencies between instructions cause several pipeline stalls (Section VI-E). Furthermore, accelerating tree walks with low level optimizations like vectorization using SIMD instructions is extremely challenging [19].

[1]The condition $x_i < v$ is the *node predicate* or the *node condition*. At each node, if the node predicate is true, then the walk moves to the left child. If not, it moves to the right child. When a leaf is reached, the value v of the leaf is returned as the prediction of the tree. The prediction of the ensemble is the sum of all tree predictions.

Today, the most popular systems for performing inference for tree-based models are libraries like XGBoost [20], LightGBM [21] and Sklearn [22]. As one would expect, these libraries implement a fixed set of optimizations for the hardware configurations they support. As hardware evolves, specializing the library to newer generations gets prohibitively expensive and usually has a high lead time. Further, these libraries do not specialize inference to the specifics of the model being used for inference.

Recent research has seen the advent of optimizing compilers for deep neural networks [23], [24], [25], [26], [27]. Such compilers have been shown to perform much better than domain-specific libraries in several cases. Surprisingly, despite their widespread usage, very few compilers exist for decision tree ensembles [11], [28]. One system, Hummingbird [11], builds on the success of DNN compilers by transforming decision tree traversals to tensor operations. However, it sometimes introduces more expensive operations like matrix multiplication just to be able to leverage compilers like TVM [24] for efficient code generation. Treelite [28], the only other compiler for tree inference, uses a simple compilation strategy. It aggressively expands all trees in the model into `if-else` statements. Both of these strategies are limited in terms of applying optimizations tailored to decision tree inference. Significant improvements over these methods are possible as we will show in our experimental evaluation.

This paper presents the design and implementation of TREEBEARD[2], an extensible optimizing compiler infrastructure for decision tree ensemble inference. TREEBEARD enables three different classes of optimizations: (i) pertaining to the nature of the algorithm, i.e., simultaneous traversal of several trees. (ii) pertaining to the properties of the tree being traversed, like imbalances in structure and probabilities of reaching leaves. (iii) targeting the characteristics of the CPU, like support for vector instructions. We carefully construct the compiler so that different optimizations are applied at different intermediate representations of the computation. By doing so, TREEBEARD facilitates the selection and composition of several optimizations at each level of abstraction, as well as automatic and efficient code generation.

We implement TREEBEARD using the MLIR compiler infrastructure [29] and evaluate it on a diverse set of decision tree-based models. We report that the code generated by TREEBEARD is significantly faster than state-of-the-art systems: it is on average 2.6×, 4.7× and 5.4× faster than XGBoost, Treelite and Hummingbird respectively on a single core. Even with a very simple parallelization strategy, TREEBEARD achieves speedups of 2.3×, 2.7× and 14× compared to XGBoost, Treelite and Hummingbird on a 16-core system.

[2]In J. R. R. Tolkein's The Lord of the Rings, Treebeard was the oldest of the Ents left in Middle-earth, an ancient tree-like being who was a "shepherd of trees".

In summary, we make the following contributions.

1) We design and implement TREEBEARD, an extensible compiler infrastructure for decision tree model inference. TREEBEARD is built to allow exploration of several optimizations and code generation techniques.
2) We propose novel tiling transformations that reduce the cost of tree walks by grouping tree nodes into "tiles". We propose tiling algorithms that can utilize specific properties of the model being compiled.
3) We design and implement various model and loop transformations that significantly improve generated inference code performance. TREEBEARD also enables parallelization of the inference computation.
4) We develop a general infrastructure for the vectorization of decision tree walks. This includes general support for code generation and memory-layout optimizations for tiled trees.

II. TREEBEARD OVERVIEW

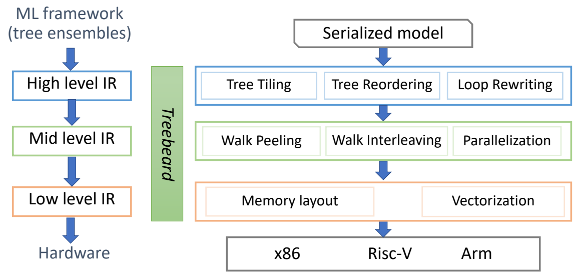
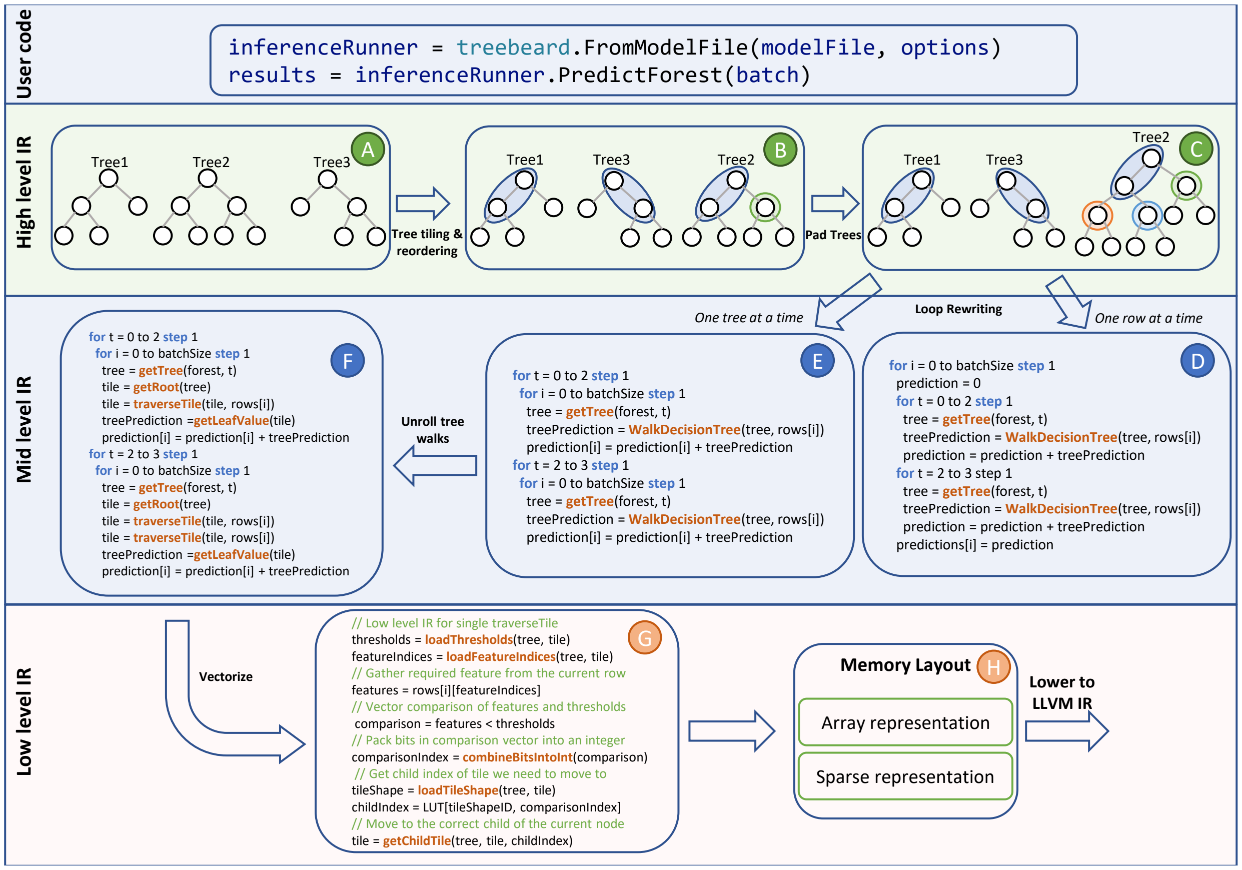


Figure 1: TREEBEARD compiler structure.

Figure 1 shows the high level structure of TREEBEARD. The input to TREEBEARD is a serialized decision tree ensemble. Given an input model our compiler generates optimized inference code. Specifically, it generates a batch inference function, `predictForest` that, given an array of input rows, computes an array of model predictions.

TREEBEARD specializes the code generated for inference by progressively optimizing and lowering a high level representation of the `predictForest` function down to LLVM IR [30]. ***Lowering*** is the process of transforming an operation at a higher abstraction to a sequence of operations at a lower abstraction. Optimizations in TREEBEARD are implemented using a combination of annotation and lowering. An operation at a higher abstraction is annotated with attributes that guide how code should be lowered. Actual transformation happens while lowering. For example, tree tiling and loop ordering are decided at the highest abstraction. These decisions are communicated to the lowering pass which explicitly encodes them in the lowered IR. Figure 2 shows



```
inferenceRunner = treebeard.FromModelFile(modelFile, options)
results = inferenceRunner.PredictForest(batch)
```

```
for t = 0 to 2 step 1
  for i = 0 to batchSize step 1
    tree = getTree(forest, t)
    tile = getRoot(tree)
    tile = traverseTile(tile, rows[i])
    treePrediction =getLeafValue(tile)
    prediction[i] = prediction[i] + treePrediction
for t = 2 to 3 step 1
  for i = 0 to batchSize step 1
    tree = getTree(forest, t)
    tile = getRoot(tree)
    tile = traverseTile(tile, rows[i])
    tile = traverseTile(tile, rows[i])
    treePrediction =getLeafValue(tile)
    prediction[i] = prediction[i] + treePrediction
```

```
for t = 0 to 2 step 1
  for i = 0 to batchSize step 1
    tree = getTree(forest, t)
    treePrediction = WalkDecisionTree(tree, rows[i])
    prediction[i] = prediction[i] + treePrediction
for t = 2 to 3 step 1
  for i = 0 to batchSize step 1
    tree = getTree(forest, t)
    treePrediction = WalkDecisionTree(tree, rows[i])
    prediction[i] = prediction[i] + treePrediction
```

```
for i = 0 to batchSize step 1
  prediction = 0
  for t = 0 to 2 step 1
    tree = getTree(forest, t)
    treePrediction = WalkDecisionTree(tree, rows[i])
    prediction = prediction + treePrediction
  for t = 2 to 3 step 1
    tree = getTree(forest, t)
    treePrediction = WalkDecisionTree(tree, rows[i])
    prediction = prediction + treePrediction
  predictions[i] = prediction
```

```
// Low level IR for single traverseTile
thresholds = loadThresholds(tree, tile)
featureIndices = loadFeatureIndices(tree, tile)
// Gather required feature from the current row
features = rows[i][featureIndices]
// Vector comparison of features and thresholds
 comparison = features < thresholds
// Pack bits in comparison vector into an integer
comparisonIndex = combineBitsIntoInt(comparison)
 // Get child index of tile we need to move to
tileShape = loadTileShape(tree, tile)
childIndex = LUT[tileShapeID, comparisonIndex]
// Move to the correct child of the current node
tile = getChildTile(tree, tile, childIndex)
```

Figure 2: TREEBEARD IR lowering and optimization details: the three abstraction levels in TREEBEARD's IR are shown. The high level IR is a tree-based IR to perform model level optimization, the mid-level IR is for loop optimizations that are independent of memory layout and the low level IR allows us to perform vectorization and other memory layout dependent optimizations.

two possible loop orders – one row at a time and one tree at a time in code snippets D and E respectively.

TREEBEARD's IR has three abstraction levels as shown in Figure 1. At the highest level (HIR), the input model is represented as a collection of binary trees. A in Figure 2 shows an example with three trees. At this level of abstraction, TREEBEARD tiles nodes together to transform a binary tree into an n-ary tree. In the example, trees in A are tiled with a tile size of 2 to produce trees in B (as indicated using colored ellipses drawn around nodes that are in the same tile).

Tree reordering (also shown in B) and padding (shown in C) are other transformations that are performed at this abstraction. Tree reordering groups identically structured trees so that they can share the same traversal code to improve locality in the instruction cache. With the tiling shown in Figure 2, Tree1 and Tree3 have depth 1, while Tree2 has a depth of 2. Therefore, the trees are reordered so that Tree1 and Tree3 are together. As can be seen in code snippets D and E in Figure 2, there are only two `WalkDecisionTree` operations while there are three trees in the model. This is possible because Tree1 and Tree3 can share the same inference code. Padding enables unrolling tree walks to remove branching overheads. In C in Figure 2, Tree2 is padded so that all leaves are at the same level in the tiled tree.

Note that all these optimizations are enabled by the domain-specific representation of HIR. Optimizations like tiling and padding would have been much harder in a traditional loop based IR or even in TREEBEARD's lower-level abstractions.

After these domain-specific optimizations are performed, mid-level IR (MIR) code is generated through lowering. At this level of the IR (code snippets D, E and F in Figure 2), the order in which tree, input row pairs are walked is explicitly represented in the IR using loop nests. D and E in Figure 2 show two possible ways that loop nests could be generated.

MIR allows optimizations that are independent of the final memory layout of the model. Operations in MIR are still abstract and leave lower-level implementation details unspecified. For example, the operation `WalkDecisionTree` represents all valid ways to compute the prediction of a decision tree given an input row. As all leaves of the tiled tree Tree2 are at depth 2, TREEBEARD completely unrolls the second `WalkDecisionTree` operation in E and implement it with exactly two `traverseTile` operations without any control flow (code snippet F). Even though frameworks like LLVM (and other standard compilers) perform loop unrolling, they cannot generate similarly optimized code because their IRs are at a much lower abstraction. Tree walk unrolling is one of several such optimizations implemented in MIR (Section IV).

TREEBEARD then further lowers the IR to explicitly represent the memory layout of the model. Buffers to hold model values are inserted into the generated code and all tree operations in the mid-level IR are lowered to explicitly reference these buffers. Additionally, at this level of the IR, TREEBEARD generates vectorized code. Code snippet G in Figure 2 shows operations for which SIMD instructions are eventually generated. Finally, the inference function is translated to LLVM IR and JIT compiled to executable code.

We implement TREEBEARD using the MLIR compiler infrastructure [29]. MLIR provides modular and reusable infrastructure to develop intermediate representations for domain-specific compilers. Firstly, to enable compiler writers to quickly design and build intermediate representations, MLIR has a large set of ***dialects*** that provide operations from various domains. Each dialect contains a logically related set of operations and types; dialect transformations and conversions allow lowering and conversion of these operations and types to other lower-level dialects and subsequently translation to LLVM IR. For example, the `scf` dialect provides control flow constructs like loops and if-else operations. The `memref` dialect provides functionality to manage buffers. We build TREEBEARD using a custom MLIR dialect and a combination of dialects provided in MLIR. Secondly, MLIR provides an extremely powerful operator rewriting infrastructure. We make extensive use of this to implement transformations.

III. OPTIMIZATIONS ON HIGH-LEVEL IR

This section describes tree tiling and tree reordering, two optimizations performed at the highest level of abstraction. Recall that at this level the `predictForest` operation is abstractly represented by a set of decision trees.

A. Notation

We represent a decision tree by $\tau = (V, E, r)$ where V is the set of nodes, E the set of edges and $r \in V$ is the root. For each node $n \in V$, we define the following.

1) $threshold(n) \in \mathbb{R}$, the threshold value for n.
2) $featureIndex(n) \in \mathbb{N}$, the feature index for n.
3) $left(n) \in V$, the left child of n or $\emptyset$ if n is a leaf. If $left(n) \neq \emptyset$, then $(n, left(n)) \in E$.
4) $right(n) \in V$, the right child of n or $\emptyset$ if n is a leaf. If $right(n) \neq \emptyset$, then $(n, right(n)) \in E$.

We use $L_\tau \subseteq V$ to denote the set of leaves.

B. Tree Tiling

While a decision tree is naturally represented as a binary tree, and can be walked with simple code, naïve implementation strategies are inefficient due to one or more of the following reasons. They (i) make poor use of the memory hierarchy, (ii) have poor instruction level parallelism due to true dependencies and (iii) cannot use vector instructions (see Section VI-E). This section proposes a tiling optimization to address some of these issues. Tiling groups multiple nodes of a decision tree into a single tile, effectively transforming a binary tree into an n-ary tiled tree. Note that an ideal decision tree traversal visits only a subset of nodes in a tile, with strict dependency between nodes at different levels. Unfortunately, this dependency prevents parallelization/vectorization of the tree walk. However, by speculatively evaluating conditions of all nodes in a tile, we can eliminate this dependency and enable the compiler to generate vectorized code (see Section V-A) to traverse trees. Secondly, tiling improves spatial locality by grouping together nodes that are likely to be accessed together (Section V-B) and enables better use of available memory bandwidth through vectorization. We present two different tiling heuristics later in this section.

Once trees are tiled TREEBEARD generates tree walks with the code structure shown below.

```
1  WalkDecisionTree(...) {
2    tile = getRoot(tree)
3    while (!isLeaf(tree, tile)) do {
4      // Evaluate predicates of all nodes in the tile
5      predicates = evaluateTilePredicates(tile, rows[i])
6
7      // Move to the correct child of the current tile
8      tile = moveToChildTile(tree, tile, predicates)
9    }
10   treePrediction = getLeafValue(tile)
11 }
```

The code above computes the prediction of decision tree `tree` for the input row `rows[i]`. It is an abstract representation of a tiled tree walk that enables efficient lowering of specific steps in subsequent stages. `evaluateTilePredicates` (speculatively) computes the predicates of all nodes in a tile (line 6)[3] on the input row `rows[i]`. Then `moveToChildTile` (line 9), uses the computed predicate values to determine which child of the current tile to move to[4]. We defer a description of how

[3]`evaluateTilePredicates` expands to the operations from line 10 to 15 in the listing in Section V-A.

[4]`moveToChildTile` expands to the operations from line 18 to 25 in the listing in Section V-A.

these operations are lowered to Section V-A and instead focus on tiling algorithms in this section.

1) Conditions for Valid Tiling: While any arbitrary partitioning of the nodes of a tree could be considered for tiling, we impose a few simple constraints to simplify the design of the compiler. Given a tree $\tau = (V, E, r)$ and a tile size n_t we impose the following constraints on the generated tiles $\{T_1, T_2, ..., T_m\}$.

Partitioning: $T_1 \cup T_2 ... \cup T_m = V$ and $T_i \cap T_j = \emptyset$ for all $i \neq j$.

Connectedness: If $u, v \in T_i$, there is a (undirected) path connecting u and v fully contained in T_i.

Leaf separation: $\forall l \in L_\tau : l \in T_i \rightarrow v \notin T_i \ \forall v \in V \backslash \{l\}$.

Maximal tiling: If there are tiles such that $|T_i| < n_t$, then there is no $v \in V \backslash \{T_i \cup L_\tau\}$ such that $(u, v) \in E$ for some $u \in T_i$.

The **partitioning** and **maximal tiling** constraints together ensure that we group nodes into as few tiles as possible. **Connectedness** ensures that each tile is a sub-tree, a natural grouping of nodes that are likely to be accessed together. The **leaf separation** constraint ensures that leaves are not grouped with internal nodes, as leaves in a decision tree need special handling and are used to check for walk termination and to determine the output (prediction). Thus, the **leaf separation** constraint ensures that tiles are homogeneous. This in-turn allows TREEBEARD to specialize the in-memory layout of trees and also simplifies code generation. We discuss leaf handling and tree layout in Section V-B. We refer to any tiling that satisfies the above constraints as a *valid* tiling.

2) Rationale for Different Tiling Methods: Before we present different tiling methods, we first discuss an interesting property of decision trees and explore whether it can be exploited to optimize tree traversal. We find that the probabilities of reaching different leaves in a tree can vary significantly, making some root to leaf paths more frequently traversed than others. As the set of decision trees that are walked for a given model is fixed, we ask whether this statistical property can be exploited to generate efficient traversal code.

To understand how often different leaves are reached, let us look at the percentage of leaves required to cover a significant part (say 80% or 90%) of the training data[5]. This is captured in Figures 3a and 3b. Each line in these graphs corresponds to a fixed fraction (say f) of the training data. A point on this line at coordinate (x, y) means that a fraction y of trees in the model could cover a fraction f of all training inputs with a fraction x of leaves. For example, the first point on the $f = 0.9$ line in Figure 3a(at the bottom left of the plot) indicates that about 52% of trees (y value) need only 1% of their leaves (x value) to cover 90% of the training input. In general, Figure 3a shows that very few leaves are needed to cover a very large fraction of inputs for the benchmark `airline-ohe`. This means that a small fraction of leaves are very likely. We call trees with a small number of extremely likely leaves ***leaf-biased***. On the other hand, for the benchmark `epsilon`, Figure 3b shows that trees need a much larger fraction of their leaves to cover a significant fraction of the training input. This means that most trees in `epsilon` are not leaf-biased. For leaf-biased trees, it would be beneficial to tile nodes such that the depth of the most probable leaves are minimized even at the expense of increasing the depth of the less probable leaves. We call the tiling algorithm designed to do this *probability-based tiling* (Section III-C).

However, when there is no clear leaf-bias, a reasonable objective is to minimize the number of nodes executed speculatively. An intuitive heuristic to achieve this objective is to minimize the depth of each tile that is constructed. *Basic tiling* (Section III-D) is a heuristic designed to do this. It is possible to define other variants of tiling(e.g., minimize the maximum leaf depth), but we leave this for future work. We believe that TREEBEARD's tiling infrastructure can support and optimize all such variants that produce valid tilings.

C. Probability-Based Tiling

The goal of probability-based tiling is to minimize the average inference latency. We observe that the latency of one tree walk is proportional to the number of tiles that need to be evaluated to reach a leaf. Therefore, the average inference latency can be minimized by minimizing the expected number of tiles that are evaluated to compute one tree prediction.

More formally, the problem is to find a *valid* (as defined in Section III-B1) tiling $\mathscr{T}$ such that the following objective is minimized.

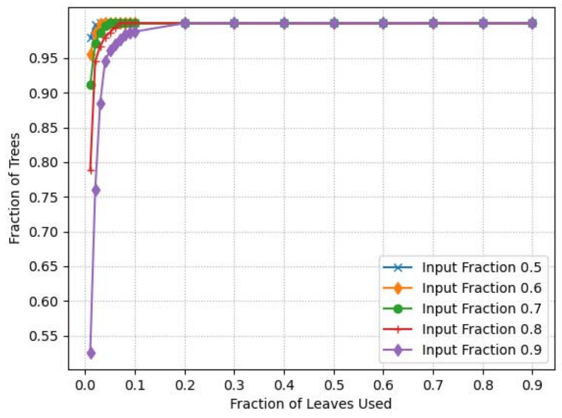
$$\min_{\mathscr{T} \in \mathscr{C}(\tau)} \sum_{l \in L_\tau} p_l.depth_{\mathscr{T}}(l)$$

where the minimization is over all valid tilings $\mathscr{T}$ of the tree τ, $depth_{\mathscr{T}}(l)$ is the depth of the leaf l given tiling $\mathscr{T}$. p_l is the probability of reaching leaf l as observed during training.

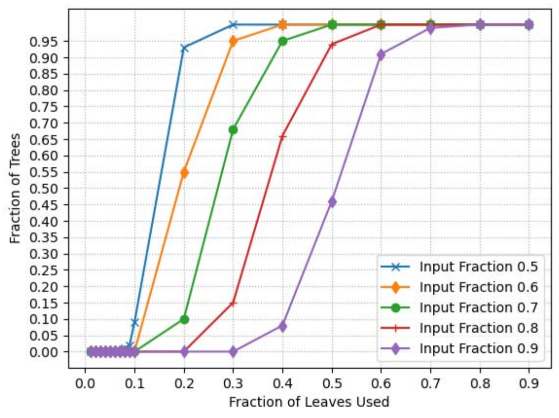
Even though the above problem can be solved optimally using dynamic programming, we use a greedy algorithm in the interest of simplicity. Algorithm 1 lists the algorithm to construct a valid tiling given the node probabilities[6]. The algorithm starts at the root and greedily keeps adding the most probable legal node to the current tile until the maximum tile size is reached. Subsequently, the tiling procedure is recursively performed on all nodes that are destinations for edges going out of the constructed tile. In the algorithm, these edges are denoted by $Out(Tile)$ and the subtree rooted at a node v is denoted by S_v. It is easy to see that the tiling constructed by Algorithm 1 is valid.

[5]We use the training data to compute tree statistics since the test data must have a distribution similar to the training data.

[6]Probabilities for internal nodes can be computed from probabilities for leaves by summing up the probabilities of all leaves that belong to the sub-tree rooted at the internal node. Leaf probabilities are collected during training.



(a) `airline-ohe`



(b) `epsilon`

Figure 3: Statistical profiles for `airline-ohe` and `epsilon`.

Algorithm 1 Greedy probability-based tree tiling.

```
1: procedure TILETREE($\tau = (V, E, r)$, $n_t$)
2:     if $r \in L_\tau$ then
3:         return $\{r\}$
4:     end if
5:     $Tile \leftarrow \{r\}$
6:     while $|Tile| < n_t$ do
7:         $e = (u, v) \in Out(Tile)$ st $p(v)$ is max and $v \notin L$
8:         if $e = \emptyset$ then
9:             break
10:        end if
11:        $Tile = Tile \cup \{v\}$
12:    end while
13:    $Tiles = \{Tile\}$
14:    for $(u, v) \in Out(Tile)$ do
15:        $Tiles \leftarrow Tiles \cup TileTree(S_v, n_t)$
16:    end for
17:    return $Tiles$
18: end procedure
```

As probability-based tiling is only beneficial for leaf-biased trees, we perform probability-based tiling on trees only when a small fraction (α) of leaves cover a large part (β) of the training inputs.

D. Basic Tiling

Algorithm 2 shows the basic tiling algorithm that produces a valid tiling. It attempts to minimize the depths of all constructed tiles. Tiling starts at the root and constructs a tile *Tile* by performing a level order traversal. The call `LevelOrderTraversal(`τ`, `n_t`)` picks the next n_t non-leaf nodes according to the standard level order tree traversal algorithm. Once the current tile is constructed, the tiling procedure is recursively performed on subtrees rooted at each node that is a destination of an edge going out of the constructed tile. It is easy to see that the tiling constructed by Algorithm 2 is valid.

Algorithm 2 Basic tree tiling

```
1: procedure LEVELORDERTRAVERSAL($\tau = (V, E, r)$, $n_t$)
2:     $queue = \{r\}$
3:     $Tile = \emptyset$
4:     while $\neg queue.empty() \wedge |Tile| < n_t$ do
5:         $n = queue.dequeue()$
6:         if $n \in L_\tau$ then
7:             continue
8:         end if
9:         $Tile = Tile \cup \{n\}$
10:        $queue.enqueue([left(n), right(n)])$
11:    end while
12: end procedure
13:
14: procedure TILETREE($\tau = (V, E, r)$, $n_t$)
15:    if $r \in L_\tau$ then
16:        return $\{r\}$
17:    end if
18:    $Tile \leftarrow LevelOrderTraveral(\tau, n_t)$
19:    $Tiles = \{Tile\}$
20:    for $(u, v) \in Out(Tile)$ do
21:        $Tiles \leftarrow Tiles \cup TileTree(S_v, n_t)$
22:    end for
23:    return $Tiles$
24: end procedure
```

One interesting property of this tiling algorithm is that it naturally reduces the imbalance in trees, especially at large tile sizes. As the algorithm traverses down to sparser levels of the tree, it naturally groups sub-trees containing chains of nodes, thus balancing the trees. While it is possible to further enhance the algorithm to explicitly balance tiled trees, we find that basic tiling suffices in practice.

E. Loop Rewriting

TREEBEARD supports a wide range of loop rewrites in the process of lowering from the high level IR to the mid level IR. TREEBEARD supports fissing and permuting of the loop nest over the tree, input row pairs. Figure 2 shows the specific case of permuting loops. Two versions of MIR generated for two different loop orders are shown – "one tree at a time" that walks one tree for all input rows before moving to the next tree (code snippet E) and "one row at a time" that walks all trees for an input row before moving to the next row (code snippet D). The structure of the loop nest to be generated in MIR is decided at the HIR level and communicated to the lowering pass through attributes (as mentioned in section II).

F. Tree Reordering

Specializing the code, as we do later in MIR for example by unrolling tree walks, for each tree in a model comes at a cost. First, the size of the generated code increases if the code generator needs to generate different code for different trees. This causes several front end stalls like instruction cache misses and delays in instruction decoding. Second, some cross tree optimizations like tree walk interleaving (applied at the lower levels of abstraction) are more effective when multiple trees share identical code.

In order to handle this, TREEBEARD groups trees by tree structure so that they can share traversal code. For example, the compiler pads trees with dummy tiles to make them balanced and then sorts the trees by their depth, so that isomorphic trees can share the same unrolled tree walk. Padding is only done for almost balanced trees (as generated by basic tiling). Additionally, the loop rewriting infrastructure described in Section III-E is used to modify the loop nest so that the same code walks all trees in a group. This is shown in D and E in Figure 2. Other metrics, like feature set commonality, could also be used to reorder trees [31]. However, we leave the exploration of such optimizations to future work.

IV. OPTIMIZATIONS ON MID-LEVEL IR

In this section, we present various tree walk optimizations performed by TREEBEARD on the mid-level IR.

A. Tree Walk Interleaving

A key bottleneck we found when we profiled code generated from tiled walks (even after vectorization) was that true dependencies between instructions were still causing a significant number of processor stalls. Performing a walk with a single input-tree pair did not provide enough independent instructions to keep the processor busy. In order to address this, TREEBEARD applies an unroll-and-jam transformation on the innermost loops of the loop nest. This has the effect of walking multiple tree and input row pairs in an interleaved fashion. This mitigates the dependency stalls by enabling scheduling of instructions from independent tree walks.

This optimization is performed in two steps. First, a pass on the mid-level IR transforms the loop structure. It unrolls the innermost loops of the loop nest a specified number of times and jams together tree walks from the different iterations. The following listing shows the mid-level IR when the inner loop over the input rows is unrolled by a factor of two and the two resulting tree walks are jammed together.

```
for t = 0 to numTrees step 1 {
  for i = 0 to batchSize step 2 {
    tree = getTree(forest, t)
    pred1, pred2 = InterleavedWalk((tree, rows[i]),
                                   (tree, rows[i+1]))
  }
}
```

Next when lowering, the operations to traverse each of the tree, input row pairs (the arguments to the `InterleavedWalk`) are interleaved. One step of the interleaved walk is listed below.

```
// ...
tile1 = tile2 = getRoot(tree)
// ...
threshold1 = loadThresholds(tree, tile1)
threshold2 = loadThresholds(tree, tile2)
featureIndex1 = loadFeatureIndices(tree, tile1)
featureIndex2 = loadFeatureIndices(tree, tile2)
feature1 = rows[i][featureIndex1]
feature2 = rows[i][featureIndex2]
pred1 = feature1 < threshold1
pred2 = feature2 < threshold2
tile1 = getChildTile(tile1, pred1)
tile2 = getChildTile(tile2, pred2)
// ...
```

B. Tree Walk Peeling and Tree Walk Unrolling

TREEBEARD splits the loop that performs a tree walk into two parts. It peels and introduces a `prologue` loop that walks down the tree a constant number of steps (for example, up to the depth of the first leaf) and then performs the rest of the tree walk in a separate loop.

Several rewrites of the peeled loop are possible. TREEBEARD completely unrolls the prologue if the peeled loop walks the tree upto the depth of the first leaf. In cases where TREEBEARD has already padded and balanced the tree (Section III-F), unrolling the prologue loop completely avoids all traversal induced branching. In the case of probability-based tiling, peeling enables specialization of leaf checks so that these checks are faster (and less general) for the most probable leaves.

C. Parallelization

Currently, TREEBEARD performs a naïve parallelization of the inference computation. When parallelism is enabled, the loop over the input rows is parallelized using MLIR's OpenMP support. TREEBEARD rewrites the mid-level IR by tiling the loop over the input rows with a tile size equal to the number of cores. As a concrete example, consider the case where we intend to perform inference using a model with four trees on a batch of 64 rows. Further, assume that we wish to parallelize this computation across 8 cores. TREEBEARD then generates the following IR:

```
parallel.for i0 = 0 to 64 step 8 {
  for i1 = 0 to 8 step 1 {
    i = i0 + i1
    prediction = 0
    for t = 0 to 4 step 1 {
      tree = getTree(forest, t)
      treePrediction = WalkDecisionTree(tree, rows[i])
      prediction = prediction + treePrediction
    }
    predictions[i] = prediction
  }
}
```

Currently, TREEBEARD does not perform other standard parallelization optimizations as they are generic and independent of the problem domain. We leave a more thorough exploration of parallelizing decision trees to future work.

V. Optimizations on Low-level IR

The optimizations performed by TREEBEARD at the low-level IR are presented in this section.

A. *Vectorization*

TREEBEARD specializes the tree walk over a tiled tree to make use of vector instructions. This specialization happens as we lower low level IR to LLVM IR. These are then converted to vector instructions in the target ISA by the LLVM JIT.

The below listing shows the low-level IR for a vectorized tree walk in detail.

```
1  // A lookup table that determines the child index of
2  // the next tile given the tile shape and the outcome
3  // of the vector comparison on the current tile
4  int16_t LUT[NUM_TILE_SHAPES, pow(2, TileSize)]
5
6  WalkDecisionTree(...) {
7    tile = getRoot(tree)
8    while (isLeaf(tree, tile)==false) do {
9      thresholds = loadThresholds(tree, tile)
10     featureIndices = loadFeatureIndices(tree, tile)
11     // Gather required feature from the current row
12     features = rows[i][featureIndices]
13     // Vector comparison of features and thresholds
14     comparison = features < thresholds
15
16     // Pack bits in comparison vector into an integer
17     comparisonIndex = combineBitsIntoInt(comparison)
18
19     // Get child index of tile we need to move to
20     tileShape = loadTileShape(tree, tile)
21     childIndex = LUT[tileShapeID, comparisonIndex]
22
23     // Move to the correct child of the current node
24     tile = getChildTile(tree, tile, childIndex)
25   }
26   prediction = getLeafValue(tile)
27 }
```

When lowering to LLVM, operations `loadThresholds` and `loadFeatureIndices` (lines 11 and 13) are lowered to use vector load instructions. Predicates are then evaluated using vector comparison instructions (lines 15 to 18). The

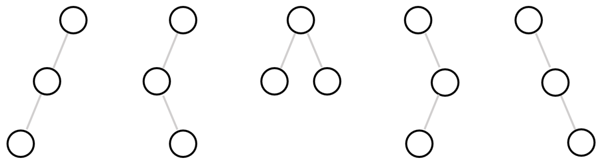


Figure 4: All possible tile shapes with tile size $n_t = 3$.

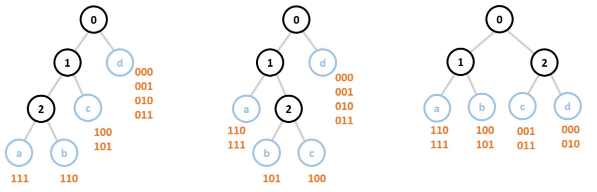


Figure 5: Example tile traversals with tile size $n_t = 3$.

next step is to determine which child to visit (lines 21 to 25). As we explain below, this depends not only on the result of the comparison but also on the shape of the sub-tree that the tile represents.

1) Tile Shapes and Tree Traversal: For a given tile size n_t, each unique legal binary tree containing n_t nodes (nodes being indistinguishable) corresponds to a ***tile shape***. Figure 4 enumerates all tile shapes with a tile size of 3.

Given the comparison vector evaluated for a tile, the tile that needs to be traversed next depends on the shape of the tile. To understand this, consider Figure 5 that shows 3 of the 5 possible tile shapes for a tile size of 3. The nodes drawn in black are members of the tile. The nodes in blue are the root nodes of the children tiles. The bit strings (written in red) show which child needs to be traversed next, given the outcomes of the comparison. The bits represent the comparison outcomes of nodes – the MSB is the predicate outcome of node 0 and the LSB the predicate outcome of node 2. For example, for the first tile shape, if the comparison outcome is 111, the next node to evaluate is *a*. It is easy to see that, depending on the tile shape, the same comparison outcomes can mean moving to different children. For example, for the outcome 011, the next tile is the 4th child (node *d*) for the first two tile shapes while it is the 3rd child for the other tile shape (node *c*)[7].

2) Lookup Table: As illustrated above a combination of tile shape and the comparison outcome can be used to determine the child tile to evaluate next. We introduce an additional map called a lookup table (LUT) to encode this information:

$$LUT : (TileShape, \{0,1\}^{n_t}) \to [1, n_t + 1] \subset \mathbb{N}.$$

[7]Children of a tile are ordered left to right regardless of depth.

The LUT is indexed by the tile shape and the comparison outcome. n_t is the tile size and $\{0,1\}^{n_t}$ is a vector of n_t booleans. The value returned by the LUT is the index of the child of the current tile that should be evaluated next. For example, if we are evaluating the first tile T_1 in Figure 5, and the result of the comparison is 110, then $LUT(TileShape(T_1), 110) = 2$ since the tile to be traversed next is the tile with node b, which is the second child of the current tile.

In order to realize this LUT in generated code, TREEBEARD associates a non-negative integer ID with every unique tile shape of the given tile size. The result of the comparison, a vector of booleans, can be interpreted as a 64-bit integer. Therefore, the LUT can be implemented as a 2 dimensional array. TREEBEARD computes the values in the LUT statically as the tile size is a compile time constant.

3) A Note on Portability: TREEBEARD's optimizations and IRs have been designed to be architecture agnostic to make it possible to reuse the same infrastructure across target architectures. The vectorization described above does not make any assumptions about the target architecture. TREEBEARD generates vector LLVM IR and leaves instruction selection entirely up to LLVM.

B. In-Memory Representation of Tiled Trees

TREEBEARD currently has two in-memory representations for tiled trees - an array based representation and a sparse representation. Both representations use an array of `structs` to represent the model.

1) Array-Based Representation: Each tree in the model is represented as an array of tiles using the standard representation of trees as arrays. The root node is at index 0 and for a node at index n, the index of its i^{th} child[8] is given by $(n_t+1)n+(i+1)$. Even though this representation is simple and efficient for small models, the memory required for bigger models is very large. This memory bloat causes performance problems due to L1 cache and L1 TLB misses. Storing leaves as full tiles (even though they represent a single value) and the empty space introduced due to the array based representation of trees that are not complete account for most of the increase.

2) Sparse Representation: We design a novel representation to reduce the large memory footprint of the array-based representation. The key features of the sparse representation are as follows.

- We add a child pointer to each tile to eliminate the wasted space in the array representation. This points to the first child of the tile. All children of a tile are stored contiguously.
- Leaves are stored as a separate array of scalar values. Across all our benchmarks, after tree tiling, a large fraction of leaves are such that all their siblings are also leaves. Such leaves are directly moved into the leaves array. For leaves for which this property does not hold, an extra "hop" is added by making the original leaf tile a normal tile. All its children are leaves with the same value as the original leaf.

[8]Nodes in the tree of tiles have n_t+1 children.

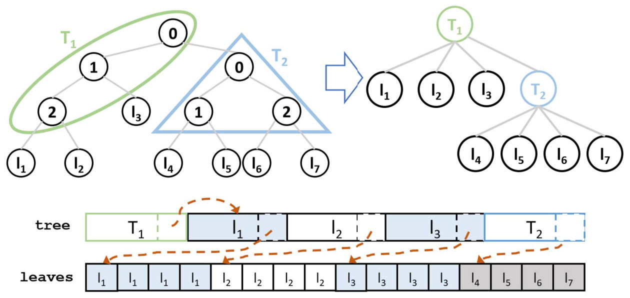


Figure 6: Sparse representation with tile size $n_t = 3$. Leaves l_4, l_5, l_6 and l_7 are moved into the `leaves` array. Extra hops are added for l_1, l_2 and l_3 as T_2 is a non-leaf tile. The new leaves added as children of l_1, l_2 and l_3 are moved to the `leaves` array.

Figure 6 shows the details of the sparse representation. The arrays depicted below show how the tree is represented in memory. The first array (`tree`) is an array of tiles and has 5 elements. Each element of the array represents a single tile and has the thresholds of the nodes, the feature indices, a tile shape ID and a child pointer (shown as red arrows). The second array, `leaves`, contains all leaf values.

As mentioned previously, the array-based representation of a tiled tree has a much higher memory footprint (8× on average) compared to a scalar representation (tile size 1). The sparse representation nearly eliminates this memory bloat. For our benchmarks, with a tile size of 8, the models stored using the sparse format are 6.8× smaller (geomean) than when stored with the array-based representation. Also, they are only 16% larger than the scalar representation.

VI. EXPERIMENTAL EVALUATION

We evaluate TREEBEARD on two machines. The first one has an Intel Core i9-11900K (Rocket Lake) processor with 8 physical cores (16 logical cores with hyperthreading), 128 GB of RAM and Ubuntu 20.04.3 LTS. The second system has an AMD Ryzen 7 4700G processor with 8 physical cores (16 logical cores with hyperthreading), 64 GB of RAM, and runs CentOS Linux release 7.9.2009. For comparisons with XGBoost, Treelite and Hummingbird, we used Python versions 3.9 and 3.10[9], XGBoost version 1.5.0, Treelite version 2.3.0 and Hummingbird version 0.4.4 running on PyTorch version 1.11 and TVM 0.9.

[9]Hummingbird benchmarks were run on Python 3.9 as Python 3.10 is not supported by Hummingbird version 0.4.4.

Dataset	#Features	#Trees	Max Depth	#Leaf-biased
abalone	8	1000	7	438
airline	13	100	9	8
airline-ohe	692	1000	9	976
covtype	54	800	9	283
epsilon	2000	100	9	0
letter	16	2600	7	0
higgs	28	100	9	8
year	90	100	9	0

Table I: List of benchmark datasets and their parameters. The column `Leaf-biased` reports the number of leaf-biased trees per benchmark with $\langle \alpha = 0.075, \beta = 0.9 \rangle$.

The benchmark models we used are trained using datasets listed in Table I. These datasets were also used by the Intel Machine Learning Benchmark suite [32]. We used the hyper-parameters from this suite for all the datasets. All models were trained with XGBoost [20]. To compute speedups, we compare the mean time taken per input row between configurations.

In order to find the best combination of optimizations, several configurations were explored for each benchmark at different batch sizes. The grid of optimizations explored is listed in Table II. We report performance results for inference performed using (i) TREEBEARD baseline code without performing any optimizations presented in Sections III, IV and V, (ii) TREEBEARD code with the combination of optimizations that performs best, (iii) XGBoost, (iv) Treelite, and (v) Hummingbird. We refer to the unoptimized TREEBEARD code as the scalar baseline.

Optimization	Configurations
Loop order	One tree at a time One row at a time
Tile size	1, 2, 4, 8
Tiling type	Basic tiling Probability-based tiling
Tree padding and unrolling	Yes, No
Tree walk interleaving	2, 4, 8
$\langle \alpha, \beta \rangle$ for leaf-bias	$\langle 0.05, 0.9 \rangle$, $\langle 0.075, 0.9 \rangle$, $\langle 0.1, 0.9 \rangle$

Table II: Space of optimizations explored.

A. Summary of Improvements on Different Hardware

Figure 7a compares the single core performance of TREEBEARD optimized code with the scalar baseline, on different hardware. The plot has two bars corresponding to the Intel and AMD machines, each showing the speedup of the optimized version over the corresponding baseline. As can be seen, TREEBEARD does consistently better on all benchmarks, attaining up to 3.5× speedup on both machines. Between Intel and AMD we find that on average the speedups are a little better on Intel. This is because the Intel machine has a much more efficient implementation of the `gather` instruction. Our vectorized implementation uses this to load features. Because of this and other differences in low level architecture we find that the best set of optimizations on the two machines are completely different. On the Intel machine we aggressively tile the tree with a tile size of 8 for all benchmarks. While on the AMD machine optimal performance is sometimes achieved at much lower tile sizes (`airline` prefers a tile size of 1, `airline-ohe`, `covtype` and `epsilon` prefer a tile size of 4). Similarly, we interleave walks more aggressively on Intel than on AMD, interleaving by a factor of eight works best on the Intel system on all benchmarks, while on the AMD one, two benchmarks perform optimally with an interleaving of four.

Figure 7b compares multi-core performance with 16 cores on different hardware. All speedups are reported with respect to an unoptimized single core run. As can be seen even with simple parallelization we are able to achieve good scalability. In fact, some optimizations combine well with parallelization to achieve super-linear speedup (see `epsilon` on AMD). This particular benchmark has the largest number of features and we suspect that on a single core it is mainly bottlenecked on the memory sub-system when accessing them.

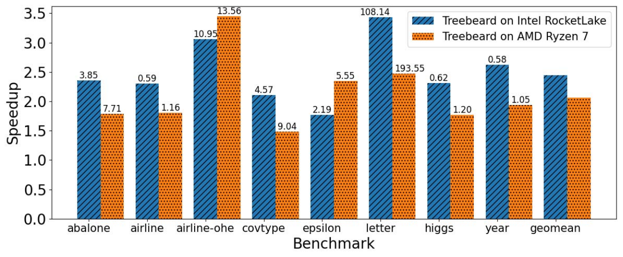
In summary by using a different set of optimizations and parameters for different benchmark-hardware combinations TREEBEARD is able to alleviate several bottlenecks and achieve a significant speedup over the baseline. TREEBEARD optimizations give a geomean speedup of 2.45× over all benchmarks on Intel Rocket Lake and 2.06× on AMD Ryzen 7.

B. Comparison with XGBoost and Treelite

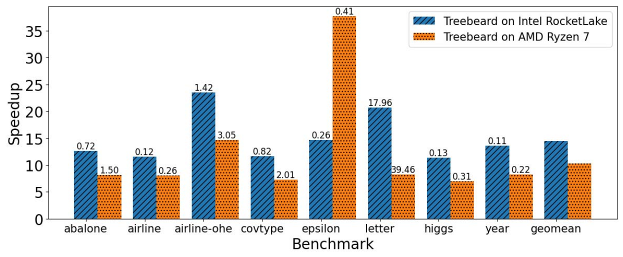
Figures 8a and 8b report a comparison of TREEBEARD with two state-of-the-art frameworks, XGBoost [20] which is widely used by ML practitioners and Treelite [28], a compiler for decision tree inference. As the plots show, TREEBEARD is significantly better than both these systems. It is at least 2× faster on most benchmarks over either system in both single-core and multi-core settings. Figure 9 shows that these performance improvements are consistent across a wide range of input batch sizes. TREEBEARD performs several novel optimizations that neither of these systems perform. In fact, apart from parallelization, all the other optimizations (tiling, tree-ordering, walk interleaving, walk peeling, vectorization and layout optimizations) in TREEBEARD are new. These results demonstrate the utility of building a generic compiler to implement different optimizations.

C. Comparison with Hummingbird

In our experiments, we find that TREEBEARD is 5.4× (geomean) faster than Hummingbird (HB) on a single-core (Figure 10) and 14× faster on 16 cores at a batch size of 1024 on the Intel Rocket Lake processor. The reasons for this high

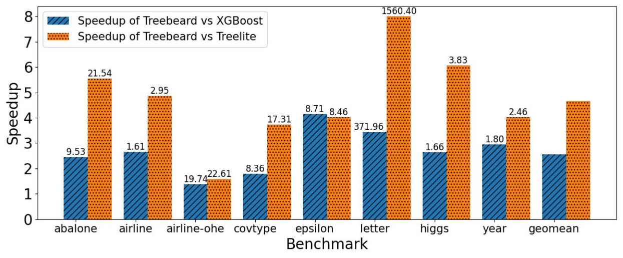


(a) Single-core

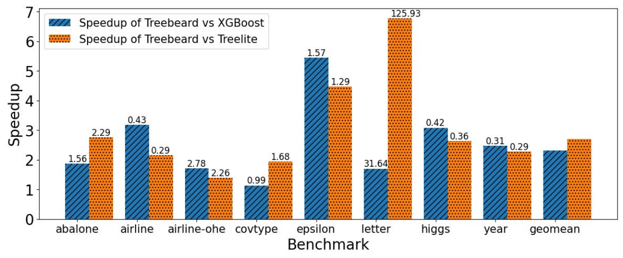


(b) 16 cores

Figure 7: Speedup of TREEBEARD optimized code over the scalar baseline at batch size of 1024. Number over each bar is the mean time (in μs) to perform inference on one input row.



(a) Single-core



(b) 16 cores

Figure 8: Comparison of TREEBEARD with XGBoost and Treelite with batch size 1024. Bars show speedup of TREEBEARD relative to XGBoost and Treelite. Numbers on bars corresponding to benchmarks are the mean inference time (in μs) per row for XGBoost and Treelite.

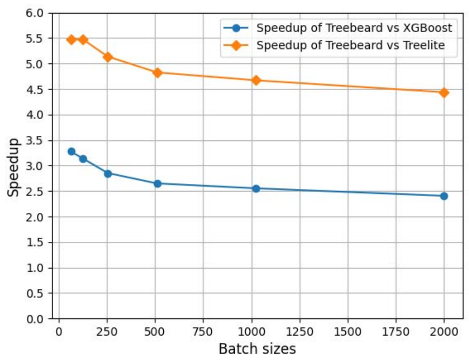


Figure 9: Geomean speedup (over all benchmarks) of TREEBEARD over XGBoost and Treelite on single-core over several batch sizes.

speedup and the difference in relative performance of HB and XGBoost compared to what was originally reported [11] are as follows.

Firstly, we note that the performance of our installation of HB is comparable[10] to what is reported in the Hummingbird paper [11]. We find HB is up to 2.4× faster than XGBoost v0.9 when run on a small number of cores (Figure 10). However, on our hardware, we find that HB scales poorly with increasing number of cores. Performance analysis using Intel VTune shows that the poor scaling is due to HB's poor average utilization of available cores (3 out of 16).

Secondly, the inference throughput of XGBoost has significantly improved in recent versions. We observe that XGBoost version 1.5 (our baseline) is on average 2.8× faster compared to version 0.9 (HB baseline). This is due to many optimizations implemented in XGBoost, including changing from a "one-row-at-a-time" loop order to "one-tree-at-a-time" order [33]. As a result, HB loses its performance advantage over XGBoost 1.5. Since TREEBEARD performs significantly better than XGBoost 1.5, the speedup relative to HB is even higher.

D. *Sensitivity Analysis*

This section reports the impact of individual optimizations and reports the sensitivity of results to both batch size and

[10]The HB experiments [11] were run on an Azure NC6 v2 machine with an older generation Intel processor while our experiments are run on a standalone server with a Rocket Lake processor. As the configurations of the two systems are different, the exact speedup numbers are different, but the trends are similar. Also, the models used in our evaluations are different from the ones used in the Hummingbird paper.

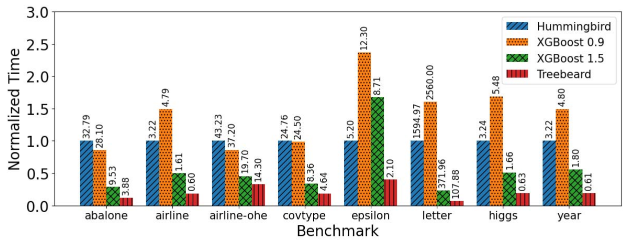


Figure 10: Single-core comparison with Hummingbird with batch size 1024. Bars show per row inference time of Hummingbird, XGBoost v0.9, XGBoost v1.5 and TREEBEARD normalized w.r.t. Hummingbird (lower is better). Numbers on the bars are the mean inference times per row in μs.

degree of parallelism. All speedup's are with respect to the scalar baseline running on a single core.

1) Impact of Optimizations: Figure 11a compares different tiling algorithms on Intel (the trends on AMD are identical, though as explained before the best tile sizes differ), the plot has two bars per benchmarks one shows the speedup over baseline when all trees use the basic tiling algorithm, while the second tiles leaf-biased trees ($\langle \alpha = 0.075, \beta = 0.9 \rangle$ as defined in Section III-C) with probability-based tiling (other trees continue to be tiled with basic tilling). The number of leaf-biased trees in each model is given in the last column of Table I. For these plots we disable all mid-level IR optimizations, but apply the low level optimizations as they go hand-in-hand with tiling.

As can be seen, even basic tiling is highly efficient, speeding up benchmarks by $1.3 - 2.5\times$. Probability-based tilling does even better. Recall that probability-based tiling makes use of additional information about leaf probabilities (how often a particular leaf is encountered while performing inference on the training data). As seen from the graph, specializing the tiling to this property of the model yields additional gains. The speedup increases from $2\times$ to $3.1\times$ for `airline-ohe`, the benchmark with the highest fraction of leaf-biased trees. Several other benchmarks see an increase in speedup of $0.2 - 0.4\times$. Three benchmarks `epsilon`, `year` and `letter` see little or no impact because these benchmarks have no leaf-biased trees.

Figure 11b shows the additional benefit over and above tiling that tree walk interleaving and tree walk peeling & unrolling achieve. As can be seen these optimizations bring in significant additional improvements (on average the speedup improves from $1.5\times$ to $2.4\times$). As discussed earlier these optimizations target different bottlenecks; while tiling improves locality and enables vectorization, these optimizations eliminate different sources of pipeline stalls. This clearly highlights the need for an extensible compiler framework where independent optimizations can be added at different levels of abstraction.

2) Impact of Batch Size: Figure 12 shows the geomean speedup over all benchmarks for various batch sizes when all TREEBEARD optimizations are performed. The graph shows that the speedups due to TREEBEARD's optimizations work equally well across all batch sizes on both the Intel and the AMD machines. We infer that it would be useful to specialize tiling over the batch size to the target machine from the slightly different trends for the Intel and AMD machines. However, we leave this for future work. We also find that performance of TREEBEARD's generated code scales well with an increasing number of cores (Figure 13). Even though TREEBEARD currently only implements a naïve parallelization strategy, performance scales reasonably well with increasing number of cores.

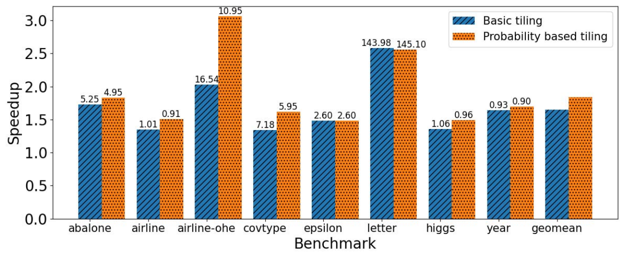
E. Microarchitectural Analysis

In this section, we report microarchitectural analysis done using Intel VTune [34]. We analyze different variants of code generated by TREEBEARD for two benchmarks, `abalone` and `higgs`, to better understand the reasons for the speedups. We analyze four code variants – (i) Scalar code processing one input row at a time (*OneRow*), (ii) Scalar code processing one tree at a time for all input rows (*OneTree*) (iii) Vector code with tile size 8 processing one tree at a time (*Vector*) and (iv) Vector code processing one tree at a time with walk interleaving and unrolling enabled (*Interleaved*).

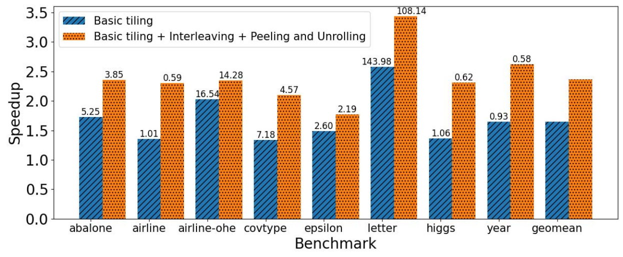
Back-end stalls are the major bottleneck for *OneRow*. For both `abalone` and `higgs`, almost three-quarters of cycles are stalls due to the back-end (73% and 71% respectively). For both benchmarks, roughly 30% of the stall cycles are due to memory and the remaining 40% are core related stalls. The variant *OneTree* better exploits the reuse of trees by walking the same tree for multiple inputs, reducing the backend stalls to 48% for `abalone` and 60% for `higgs`, with all the gains coming from a reduction in memory stalls. We note that this loop interchange optimization was recently implemented in XGBoost and resulted in a significant performance improvement [33].

Next, we analyze the reasons for the speedup of *Vector* and *Interleaved* over *OneTree*, which also explains the speedups TREEBEARD provides over XGBoost. The variant *Vector* includes tree tiling and vectorization and is on average $1.65\times$ faster than variant *OneTree*. There are two reasons for this speedup. Firstly, there is a significant reduction in dynamic instruction counts for both `abalone` and `higgs` (about $1.7\times$ for both benchmarks) going from *OneTree* to *Vector*. Secondly, since our tree tiling strategy uses vector loads to load tile thresholds and tile feature indices, it utilizes the available L1 cache bandwidth much more efficiently. This ensures that the redundant work done in processing an entire tree tile does not increase inference time.

The *Vector* variant still has a significant number of core stalls (around 40% of execution cycles for both benchmarks). This is primarily due to data dependencies. Tree walk



(a) Tiling



(b) Walk Unrolling, Walk Interleaving

Figure 11: Impact of individual TREEBEARD optimizations at batch size 1024 (Intel Rocket Lake). Number on the bar is the average inference time per row in μs.

interleaving (Section IV-A) is an optimization designed to address this by rescheduling tree walks. Accordingly, the *Interleaved* variant has significantly fewer core stalls (28% for both benchmarks) and improved (execution) port utilization. Also, the unrolling performed in this strategy (i) reduces the loop control overhead instructions, reducing the dynamic instruction count by about a third, and (ii) increases the basic block size, enabling LLVM to generate better code.

Additionally, we profile code generated by Treelite for `abalone` and `higgs` in order to better understand the microarchitectural properties of code where trees are expanded into `if-else` statements. We find that these implementations are completely front-end bound (88% of cycles for `abalone` and 70% for `higgs` are front-end stalls). For both benchmarks, roughly half of the stall cycles are I-cache misses and the other half are due to branch mispredictions. Optimizing code generated using this strategy is much harder than the loop based code generated by TREEBEARD. In contrast, the fully optimized code generated By TREEBEARD limits front-end stalls to less than 10% for both benchmarks. These are dominated by front-end bandwidth related stalls with no I-cache misses or branch mispredictions.

Overall, our results show that TREEBEARD provides significant performance gains compared to existing systems like XGBoost, Hummingbird and Treelite. Also, these speedups are robust to changes in hardware and parameters like batch size.

VII. RELATED WORK

While decision trees are heavily used there is little prior work on building optimizing compilers for tree-based inference. We discuss below related work and contrast it to TREEBEARD.

Decision Tree Ensemble Compilers: Treelite [28] is a model compiler for decision tree ensembles and is the system most closely related to our work. Even though Treelite compiles ensembles, it only supports generation of simple `if-else` style code. It does not perform the rich optimizations TREEBEARD performs and is not easily extensible. Hummingbird [11] compiles traditional ML models to make use of tensor primitives so that they can be integrated into tensor-based frameworks like TensorFlow [35]. While Hummingbird does compile decision tree based models to both CPUs and GPUs, the primary aim of the system is to leverage progress in tensor compilers by representing ML models as tensor operations. Consequently, Hummingbird

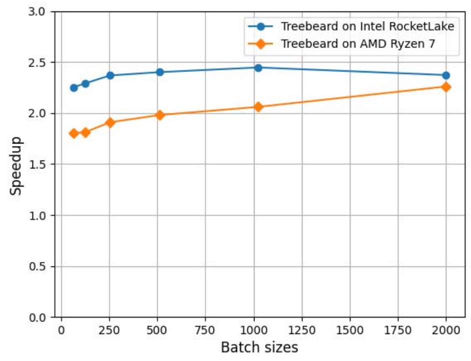


Figure 12: Single-core geomean speedup (over all benchmarks) of optimized TREEBEARD code over scalar baseline.

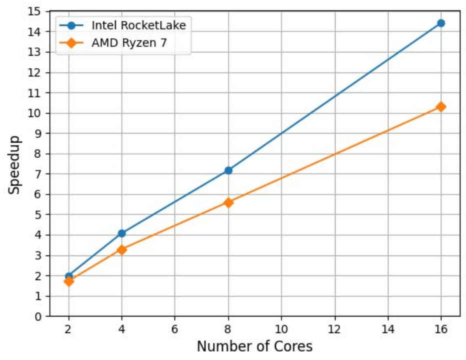


Figure 13: TREEBEARD scaling with varying number of cores. Speedup is computed over scalar baseline.

does not perform any optimizations tailored to decision trees. It also does not perform model specific optimizations that are enabled by TREEBEARD's abstractions.

Libraries: Currently, the most popular systems for decision tree-based models are libraries. XGBoost [20] and LightGBM [21] are the most popular gradient boosting libraries while scikit-learn [22] is extremely popular for random forest models. These libraries implement both training and inference. However, as mentioned in Section I, porting optimizations in these libraries to newer architectures and maintaining them is extremely difficult. Additionally, the inference routines in these libraries have to be general and cannot be specialized to a specific model.

Asadi et. al. [36] optimize tree walks by hiding dependency stalls by interleaving tree walks. In contrast, TREEBEARD is an extensible optimizing compiler that carefully implements this and many other optimizations at different levels of abstraction. QuickScorer [37], [38] is an algorithm that uses bit manipulation to compute tree predictions. Even though QuickScorer is extremely fast for smaller models, it does not scale well to larger models [39]. The goal of QuickScorer is orthogonal to the goals of TREEBEARD and the QuickScorer algorithm can easily be integrated into TREEBEARD as another traversal strategy for the system to explore. Tang et. al. [40] and Jin et. al. [41] build models to predict cache performance of decision tree ensembles on CPUs. This work is again orthogonal to the work described in this paper. Some systems have been proposed to parallelize decision tree training on CPUs and GPUs [42], [43].

Tahoe [31] is a library and a performance model for high performance tree inference on GPUs. There are several ways in which TREEBEARD and Tahoe are different. Firstly, Tahoe only chooses one of four predefined implementation strategies. TREEBEARD, on the other hand, can explore the whole space of implementation options because of its code generation approach. Second, as Tahoe uses a library-based approach, it does not generate code tailored to a model like TREEBEARD does by performing optimizations like tree walk unrolling and interleaving. Finally, all of Tahoe's optimizations are designed to address GPU specific problems. For example, it uses an elaborate and opaque heuristic based on locality sensitive hashing to coalesce accesses to the GPU global memory. We believe that the tree tiling infrastructure described in this paper will allow us to deterministically coalesce accesses to GPU global memory. While compiling decision tree inference to GPUs presents a distinct set of challenges, we believe, we can achieve this while reusing much of TREEBEARD's current infrastructure. Specifically, we expect that most of the HIR and MIR optimizations described in this paper will carry over directly while LIR optimizations and in-memory representations will need to be retargeted. This can be the subject of a separate future work.

Other Systems and Techniques: Ren et. al. [44] build an intermediate language and VM to enable SIMD execution of decision tree inference. The SIMD execution itself is implemented by hand in the VM and the VM needs to be reimplemented for every supported target architecture. Additionally, even though they perform layout optimizations, their system does not perform any model specific optimizations. Jo et. al. [19] describe techniques to vectorize tree-based applications. They do not study optimizations specific to decision trees. Both these works vectorize tree walks by performing different tree walks on each vector lane. The main issue with this approach is the divergence of tree walks. Another issue is that memory accesses are `gather`'s rather than vector loads. TREEBEARD's approach to vectorization solves both these issues. Also, these approaches are not precluded by TREEBEARD's tiling based vectorization. Multiple tiled tree walks can be combined into a single vectorized walk. We leave an exploration of this to future work. FAST [18] is a system that accelerates tree structured index search on CPUs and GPUs. FAST defines a layout for the index tree that enables vectorization of the tree walk. FAST uses a tree tiling approach to vectorize tree walks. However, FAST only uses a single triangular tile shape. TREEBEARD's basic tiling algorithm is a generalization of the tiling used in FAST. If given a perfectly balanced tree, the basic tiling algorithm would return exactly the tiling used by FAST. Also, the tree walks in FAST are hand coded using intrinsics on the CPU and CUDA on the GPU and therefore need repeated effort to implement on each target. Inspector-executor systems [45], [46] have been developed to parallelize tree walks. However, these techniques are not a good fit for decision tree inference. In decision trees, the individual node predicates are very simple and the overhead of an inspector-executor system would be prohibitive.

Code Generation Systems from Other Domains: Several compilers and code generation techniques exist for other domains. TVM [24], Tiramisu [26], CORTEX [47] and Tensor Comprehensions [25] are domain specific compilers for deep learning models. CORTEX transforms recursive computations in DNNs to loop based computations and optimizes them. However, the model properties CORTEX assumes to optimize generated code do not hold in the context of decision tree inference. For example, CORTEX assumes that control flow depends purely on data structure connectivity. This is not the case with decision trees. Control flow is determined by the input value and not by the structure of the tree. This makes the problems addressed by TREEBEARD very different from the ones handled by CORTEX.

Halide [23] is a DSL and compiler primarily designed for image processing applications. Several systems that generate optimized processing routines have also been designed for BLAS [48], [49], [50] and signal processing [51], [52]. BLIS [48] and ATLAS [49] are systems to instantiate high performance BLAS routines on multiple target architectures. CUTLASS [50] provides building blocks in the form of C++ templates to quickly instantiate high performance

BLAS functions on different GPU architectures. SPIRAL [52] is a domain specific compiler for signal processing applications that instantiates high performance routines for linear transformations like FFTs and FIR filters. FFTW [51] is a fast fourier transform compiler that generates high performance FFT routines by customizing the routine based on FFT size and the target machine. However, no such frameworks for decision tree ensembles exist currently. TREEBEARD is a first step in this direction and unlocks several future optimization opportunities.

VIII. Conclusions

This paper presents TREEBEARD, a compiler that automatically generates efficient code for decision tree inference. TREEBEARD gradually lowers inference code to LLVM IR through multiple intermediate abstractions. It composes several novel optimizations to specialize inference code for each model on each supported CPU target. Experimental evaluation demonstrates that TREEBEARD is significantly faster than state-of-the-art frameworks XGBoost, Treelite and Hummingbird.

TREEBEARD is a first step in automatically generating high performance inference code for decision tree ensembles. It significantly simplifies retargeting high performance inference routines to new hardware targets. We believe this methodology can be extended to benefit a richer ensemble of ML models.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments and feedback. Their suggestions helped improve our paper significantly. We would also like to thank the anonymous artifact reviewers for evaluating our artifact and providing feedback. We would like to thank National Instruments (NI) India for providing financial support to the first author.

Appendix

A. Abstract

This appendix describes the TREEBEARD artifact. We describe how to obtain the complete TREEBEARD source code, build it, run functional tests and performance tests. Apart from the complete source code for the TREEBEARD compiler, the artifact has scripts to build, test and run our compiler along with all benchmark models and test data that was used in our evaluation.

B. Artifact check-list

- **Algorithm:** Decision tree compilation.
- **Program:** MLIR, LLVM, Python, XGBoost, Treelite.
- **Compilation:** clang v10.0, lld v10.0, gcc v9.3.0, g++ v9.3.0.
- **Transformations:** Lowering and optimizing passes implemented in MLIR.
- **Binary:** Will be built.
- **Model:** Pre-trained XGBoost models for all datasets, based on parameters in the Intel ML Benchmark Suite [32].
- **Data set:** abalone, airline, airline-ohe, covtype, epsilon, letter, higgs, year.
- **Hardware:** Intel Core i9-11900K (Rocket Lake) processor and 128 GB of RAM.
- **Execution:** Through provided python and shell scripts.
- **Metrics:** Execution time and speedup.
- **Experiments:** A comparison between the per sample processing time of XGBoost, Treelite and TREEBEARD at a batch size of 1024 for all benchmarks in both a single-core and multi-core setting.
- **How much disk space required (approximately)?:** 15GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 2 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Workflow framework used?:** Python, shell scripts
- **Archived:** Yes DOI:10.6084/m9.figshare.20474229

C. Description

1) How to access: The complete TREEBEARD source code is available at https://github.com/asprasad/treebeard. The repo also contains instructions to build and run the code.

2) Hardware dependencies: Our evaluation was performed on a machine with an Intel Core i9-11900K (Rocket Lake) processor and 128 GB of RAM. This processor will be needed to obtain results identical to what is reported in the paper. However, if this is not possible, we expect comparable results with very recent Intel processors that have good AVX implementations.

3) Software dependencies: We used the following software for our experiments - Ubuntu 20.04.3, gcc v9.3.0, g++ v9.3.0, clang v10, lld v10, cmake v3.16.3, ninja v1.10.0, Anaconda (3-2021.11), bash, git.

4) Data sets: We used the following data sets to train models – `abalone`, `airline`, `airline-ohe`, `covtype`, `epsilon`, `letter`, `higgs`, `year`. All trained models are available in the github repo.

D. Installation

We've written several shell scripts to simplify building and executing TREEBEARD. The full set of installation instructions are provided here : https://github.com/asprasad/treebeard.

E. Experiment workflow

The artifact includes functionality tests as well as performance tests for TREEBEARD. Below, we describe how to run these tests. This section gives an overview of the tests and briefly describes the expected results. Detailed instructions to run these tests are at https://github.com/asprasad/treebeard.

1) Functionality tests: TREEBEARD has three sets of functionality tests.

- **Python functionality tests:** Executing the python script `run_python_tests.py`, after setting up the TREEBEARD python environment, will run several functionality tests that exercise the TREEBEARD python API. Each of these tests will print `pass` or `fail` on the console.

- **Treebeard sanity tests:** Running the built TREEBEARD binary with the switch `--sanityTests` will run a small set of unit and integration tests. A pass/fail result will be printed on the console at the end of the test suite.
- **[Optional] Treebeard test suite:** Running the built TREEBEARD binary (`treebeard`) will run the complete test suite. This will take about an hour to run and will print a pass/fail result on the console when all tests have completed.

2) Performance tests: The TREEBEARD repo contains two python scripts

- `treebeard_serial_benchmarks.py` and
- `treebeard_parallel_benchmarks.py`

to compare the performance of code generated by TREEBEARD with XGBoost and Treelite in both a single-core setting and a multi-core setting. Running these scripts will generate the data that was used to plot Figures 8a and 8b.

F. Evaluation and expected results

1) Functional testing: All tests mentioned in Section E1 should pass.

2) Performance testing: Running the python scripts mentioned in Section E2 will print (to stdout) a table of values in the CSV format. The table contains the following data.

1) Inference time per sample (in seconds) for TREEBEARD, XGBoost and Treelite.
2) Speedup of TREEBEARD relative to XGBoost and Treelite for each benchmark.
3) Geomean speedup of TREEBEARD over all benchmarks relative to XGBoost and Treelite.

This data is used to plot the graphs in Figures 8a and 8b.

G. Experiment customization

The benchmark scripts listed above are hard-coded to use a TREEBEARD configuration that is tuned for the Intel processor on which we ran our experiments. Running the benchmark scripts with the `--explore` switch will explore a few other predefined configurations and find the best one for the machine on which code is being executed. However, this will mean that the python script will take significantly longer to complete.

REFERENCES

[1] "Kaggle state of data science and machine learning 2021," https://www.kaggle.com/kaggle-survey-2021, accessed: 2022-04-16.

[2] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer, "Data science through the looking glass and what we found there," 2019. [Online]. Available: https://arxiv.org/abs/1912.09536

[3] S. Kotsiantis, "Decision trees: A recent overview," *Artificial Intelligence Review*, pp. 1–23, 04 2013.

[4] D. Cossock and T. Zhang, "Statistical analysis of bayes optimal subset ranking," *IEEE Transactions on Information Theory*, vol. 54, no. 11, pp. 5140–5154, 2008.

[5] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. S. Khudia, J. Law, P. Malani, A. Malevich, N. Satish, J. M. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. M. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *CoRR*, vol. abs/1811.09886, 2018. [Online]. Available: http://arxiv.org/abs/1811.09886

[6] D. Delen, C. Kuzey, and A. Uyar, "Measuring firm performance using financial ratios: A decision tree approach," *Expert Systems with Applications*, vol. 40, p. 3970–3983, 08 2013.

[7] A. Azar and S. El-Metwally, "Decision tree classifiers for automated medical diagnosis," *Neural Computing and Applications*, vol. 23, pp. 2387–2403, 11 2013.

[8] J. Soni, U. Ansari, D. Sharma, and S. Soni, "Predictive data mining for medical diagnosis: An overview of heart disease prediction," *International Journal of Computer Applications*, vol. 17, pp. 43–48, 03 2011.

[9] V. Lalchand, "Extracting more from boosted decision trees: A high energy physics case study," 2020. [Online]. Available: https://arxiv.org/abs/2001.06033

[10] "The total cost of ownership of amazon sagemaker," https://pages.awscloud.com/rs/112-TZM-766/images/Amazon_SageMaker_TCO_uf.pdf, 2020.

[11] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 899–917. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/nakandala

[12] S. Daghaghi, N. Meisburger, M. Zhao, Y. Wu, S. Gobriel, C. Tai, and A. Shrivastava, "Accelerating slide deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more," *ArXiv*, vol. abs/2103.10891, 2021.

[13] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on CPUs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1025–1040. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/liu-yizhi

[14] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[15] "Evaluating boosted decision trees for billions of users," https://engineering.fb.com/2017/03/27/ml-applications/evaluating-boosted-decision-trees-for-billions-of-users/, 2017.

[16] "Xgboost machine learning challenge winning solutions," https://github.com/dmlc/xgboost/tree/master/demo#machine-learning-challenge-winning-solutions, 2022.

[17] "Xgboost machine learning challenge winning solutions," https://github.com/microsoft/LightGBM/blob/master/examples/README.md#machine-learning-challenge-winning-solutions, 2022.

[18] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 339–350. [Online]. Available: https://doi.org/10.1145/1807167.1807206

[19] Y. Jo, M. Goldfarb, and M. Kulkarni, "Automatic vectorization of tree traversals," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. IEEE Press, 2013, p. 363–374.

[20] 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

[21] 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," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 3149–3157.

[22] "scikit-learn : Machine learning in python," https://scikit-learn.org/stable/, accessed: 2022-04-16.

[23] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[24] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[25] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018. [Online]. Available: https://arxiv.org/abs/1802.04730

[26] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 193–205.

[27] G. Inc, "XLA (Accelerated Linear Algebra) for TensorFlow," 2017, https://www.tensorflow.org/performance/xla/.

[28] "Treelite : model compiler for decision tree ensembles," https://treelite.readthedocs.io/en/latest/, accessed: 2022-04-16.

[29] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[30] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[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] "Intel machine learning benchmarks," https://github.com/IntelPython/scikit-learn_bench, 2020.

[33] "Xgboost predict improvement," https://github.com/dmlc/xgboost/pull/6127, 2020.

[34] "Intel vtune profile," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.40d3ea, 2022.

[35] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[36] N. Asadi, J. Lin, and A. P. de Vries, "Runtime optimizations for tree-based machine learning models," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2281–2292, 2014.

[37] C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 73–82. [Online]. Available: https://doi.org/10.1145/2766462.2767733

[38] ——, "Exploiting cpu simd extensions to speed-up document scoring with tree ensembles," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 833–836. [Online]. Available: https://doi.org/10.1145/2911451.2914758

[39] S. Buschjager, K.-H. Chen, J.-J. Chen, and K. Morik, "Realization of random forest for real-time evaluation through tree framing," in *2018 IEEE International Conference on Data Mining (ICDM)*, 2018, pp. 19–28.

[40] X. Tang, X. Jin, and T. Yang, "Cache-conscious runtime optimization for ranking ensembles," in *Proceedings of the 37th International ACM SIGIR Conference on Research amp; Development in Information Retrieval*, ser. SIGIR '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1123–1126. [Online]. Available: https://doi.org/10.1145/2600428.2609525

[41] X. Jin, T. Yang, and X. Tang, "A comparison of cache blocking methods for fast execution of ensemble-based score computation," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 629–638. [Online]. Available: https://doi.org/10.1145/2911451.2911520

[42] K. Jansson, H. Sundell, and H. Boström, "gpurf and gpuert: Efficient and scalable gpu algorithms for decision tree ensembles," *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pp. 1612–1621, 2014.

[43] A. Nasridinov, Y. Lee, and Y.-H. Park, "Decision tree construction on gpu: ubiquitous parallel computing approach," *Computing*, vol. 96, pp. 403–413, 2013.

[44] B. Ren, T. Mytkowicz, and G. Agrawal, "A portable optimization engine for accelerating irregular data-traversal applications on simd architectures," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, jun 2014. [Online]. Available: https://doi.org/10.1145/2632215

[45] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 12–25. [Online]. Available: https://doi.org/10.1145/1993498.1993501

[46] 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

[47] P. Fegade, T. Chen, P. B. Gibbons, and T. C. Mowry, "Cortex: A compiler for recursive deep learning models," 2020. [Online]. Available: https://arxiv.org/abs/2011.01383

[48] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, jun 2015. [Online]. Available: https://doi.org/10.1145/2764454

[49] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[50] "Nvidia cutlass," https://github.com/NVIDIA/cutlass, accessed: 2022-04-16.

[51] M. Frigo, "A fast fourier transform compiler," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 169–180. [Online]. Available: https://doi.org/10.1145/301618.301661

[52] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.