2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT) | 979-8-3503-4254-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/PACT58117.2023.00023

Accelerating Decision-Tree-based Inference through Adaptive Parallelization

Jan van Lunteren
IBM Research Europe
Rüschlikon, Switzerland
jvl@zurich.ibm.com

***Abstract*—Gradient-boosted trees and random forests are among the most popular machine learning algorithms. They are applied to a broad spectrum of applications as diverse as search engine ranking, credit card fraud detection, customer relationship management, fault diagnosis in machinery, and geological studies, to name a few. Inference of these models is supported by many widely-used libraries and frameworks including Scikit-Learn, XGBoost, and ONNX Runtime.**

Many of the inference algorithms integrated in these libraries are optimized for performing fast predictions for large input batches, often targeted at ensemble models containing relatively shallow decision trees. This does not necessarily match well with the requirements of new emerging applications that depend on real-time predictions for individual samples or small input batches. Also, cloud-based inference services put more emphasis on small memory footprints.

This paper aims to fill this gap by proposing a novel inference scheme for decision-tree ensembles that efficiently handles a wider range of application requirements and model characteristics. Performance is maximized for an extensive number of parameter combinations through a new concept in which a predict function is selected dynamically at runtime. This is done in a way that the processing resources, including the use of SIMD vector processing units and multithreading, are optimally exploited. Experiments have shown substantial performance improvements over state-of-the-art algorithms for common models.

***Index Terms*—decision trees, random forests, machine learning, parallel processing, multithreading**

I. Introduction

Decision trees in machine learning have a long history. Automatic Interaction Detection (AID) [1] and THeta Automatic Interaction Detection (THAID) [2] are often considered to be the first published decision tree algorithms for regression and classification, respectively, which were combined and extended into the Classification And Regression Trees (CART) algorithm [3]. Later, ensemble learning methods such as *bagging* [4], *gradient boosting* [5]–[7] and *random forests* [8], [9] were developed, which combine multiple decision trees to substantially improve prediction accuracy over individual decision trees. Despite this long history, decision trees and ensemble methods such as random forests and gradient boosting are still among the most frequently used machine learning algorithms today, as shown by a recent Kaggle survey [10]. This is because of the inherent advantages coming from the relatively simple concept, support for numerical and categorical features and interpretability of predictions.

Although basic research on decision trees has existed for decades, the widespread deployment of machine learning in recent years has boosted research into innovative approaches to further increase performance. This research was initially centered around training, but inference has gradually become more important as well, as can be seen from the increase in related publications in this field [11]–[20]. Originally, these efforts were focused on the inference of larger input batches, for which the prediction latency of an individual input sample was less important and processing delays could be amortized over the entire batch. This has fundamentally changed with the emergence of new applications that rely on real-time online predictions involving only one or a few input samples. Such applications are typically encountered in the financial services sector and include the scoring of credit card transactions for fraud detection as well as real-time anti-money laundering operations, for example. Low latency operation is critical because the inference operation is part of a real-time processing pipeline, in which individual transactions or small batches of transactions are processed on-the-fly. In addition, small memory footprints are important in shared, multi-user applications, e.g., in cloud serving, where multiple models may simultaneously be held in memory to support concurrent model inference requests.

This paper addresses these challenges in multiple ways. A first contribution consists of optimized versions of conventional breadth-first and depth-first decision tree traversal algorithms that enable efficient use of SIMD vectorization and the exploitation of node-level access probabilities to speedup the processing of both shallow and deep tree structures. This is in contrast to state-of-the-art schemes, where the use of SIMD vectorization is typically limited to shallow trees and is not combined with optimizations based on node-level access characteristics. A second contribution enables efficient inference for individual samples and small to large input batches through a novel concept in which a collection of predict functions is designed, with each function implementing a different combination of parallelization using SIMD vectorization and multithreading. The most suitable and performant predict function is then selected dynamically during inference based on model, request and platform parameters. To the best of our knowledge, this is the first time that such a concept has been proposed for decision tree inference.

979-8-3503-4254-3/23/$31.00 ©2023 IEEE
DOI 10.1109/PACT58117.2023.00023

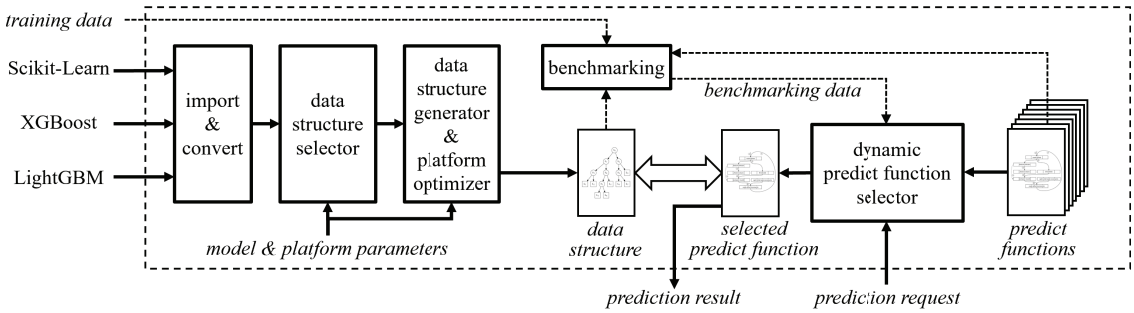

Fig. 1. Block diagram of inference function.

The remainder of the paper is organized as follows. Section II describes the design of the proposed inference function. Section III discusses related work. Optimized data structures for breadth-first and depth-first tree structures are presented in Section IV. Sections V and VI describe how the tree traversal algorithms for those data structures can be accelerated using SIMD vectorization and multithreading, respectively. Section VII presents experimental results and a performance comparison with other state-of-the-art schemes and Section VIII concludes the paper.

II. Design Overview

Fig. 1 shows a block diagram of the CPU-based decision-tree inference function that is presented in this paper. It supports importing random forest and gradient boosting models trained in Scikit-Learn [21], XGBoost [22] and LightGBM [23] and exported to PMML [24], ONNX [25] or selected proprietary formats. Imported models are first converted into an internal decision-tree ensemble representation that is used as input to a data structure selector, which selects a data structure type out of two candidates (the *OBF* and *ODF* structures that will be introduced in Section IV) based on model and platform parameters. The selected data structure is then generated and optimized for the given platform, taking into account cache sizes, the availability of SIMD instructions, and other platform characteristics. For each data structure type, multiple predict functions have been implemented, involving different combinations of SIMD vectorization and multithreading, as will be discussed in Section V. At inference time, the predict function that is expected to perform best is selected based on prediction request parameters, in particular the batch size and the number of CPU threads available for the given prediction request. None of the steps shown in Fig. 1 changes the basic functionality of the imported model, and, consequently, the inference function will generate exactly the same prediction results as if the original model was scored in its native runtime, i.e., Scikit-Learn, XGBoost or LightGBM.

The selection of the best data structure and predict function involves complex dependencies on platform parameters (e.g., cache sizes, SIMD instructions), model parameters (e.g., tree counts and depths), and prediction request parameters which renders the creation of general selection rules for the above data structure selector and dynamic predict function selector impractical. Therefore, instead a benchmarking function is applied, as shown in Fig. 1, which, using training data or other data serving this purpose, evaluates the performance of the various predict functions for both data structure types and for a range of prediction request parameters. This information is then used by the data structure selector and the dynamic predict function selector. If no data is available for the benchmarking operation, then the selection will be based on previously collected benchmarking data for general models. Ongoing research investigates the possibility of training a machine learning model for data structure and prefix function selection, however, this is outside the scope of this paper.

The inference function is implemented in C++ code as part of an extension module that can be accessed in Python using a Scikit-Learn-like interface.

III. Related work

The importance of random forests and gradient boosting models has triggered a considerable amount of research in recent years to accelerate the inference of decision-tree-ensemble models on a range of platforms, including CPUs, GPUs [11], [26], FPGAs [12], CPUs with neural network hardware accelerators [13], and memristor-based memory [27].

Because of the scope of this paper, this section will focus on CPU-based inference, which can be roughly divided into 1) methods that translate the decision-tree ensemble into compilable code with the decision-tree-node comparisons directly mapped on if-then-else statements or predicates, and 2) methods that generate internal data structures from the decision-tree definitions that are processed by fixed functions to perform the inference. Examples of the first category include TreeLite [14], VPRED [28], an optimized version of VPRED using cache-blocking [29], lleaves [30], and the optimized 'if-else' trees presented in [15], [16].

Scikit-Learn [21], XGBoost [22], LightGBM [23], ONNX Runtime [31] and the *native* tree discussed in [15] belong to the second category and implement more conventional algorithms based on breadth-first and depth-first traversal of the decision trees. QuickScorer [17], [32] takes a different

approach that is based on processing feature comparisons related to multiple split nodes throughout the tree ensemble in a first step, followed by determining the actual traversed paths through the trees and the overall prediction result from these processing results that are encoded as bit vectors and are operated on using logical bitwise instructions. This is fundamentally different from conventional tree-traversal algorithms that only process tree nodes when they are on a currently traversed path through a tree. V-QuickScorer [33] is an improved version of QuickScorer exploiting SIMD extensions. RapidScorer [19] can be regarded as a further improvement of QuickScorer removing some of its limitations related to deeper trees. The tree tiling technique proposed as part of the TreeBeard compiler [20] is also based on performing the feature comparisons for all nodes within a given tree tile in parallel using SIMD instructions, from which the traversed paths are then determined. Tree tiles are, however, located within individual trees, and different data structures are used.

Besides the above schemes that process a decision-tree ensemble model 'as is', other optimizations have been proposed that optimize performance by limiting or omitting part of the processing. One example are 'early exit' schemes [34] [18], which do not process all trees in an ensemble to improve inference performance by compromising slightly on accuracy. Another example are *oblivious* decision trees such as CatBoost [35] and others [36], [37] which are constrained to use identical node comparisons involving the same input features and thresholds for split nodes at the same tree depth.

Many of the above methods convert the decision trees into *perfect trees* of the same size. These are trees in which each split node has two child nodes and all leaf nodes are at the same level, resulting in identical path lengths between the root node and any leaf node. This enables efficient exploitation of SIMD instructions to parallelize multiple tree traversals by performing these in lock-step with all traversals ending at the same time. In order to keep the processing steps simple for SIMD exploitation, typically no node-level optimizations are performed to improve spatial locality properties. The main disadvantage of perfect trees is the exponential growth of the number of nodes as a function of the tree depth, which limits its application to shallow trees (e.g., with a maximum depth up to around eight) to prevent a tree-size 'explosion'. Although perfect trees can work efficiently for gradient boosting models that usually involve more shallow trees, this may not always apply to random forests which can involve deeper tree structures [16].

When comparing the above related work with the approaches proposed in this paper, the following are notable. In the next section, we will describe two optimized tree traversal algorithms, *OBF* and *ODF*, based on conventional breadth-first and depth-first tree traversal concepts, that have been extended to support SIMD vectorization and node-level access probability exploitation (*ODF*) for both shallow and deep trees. Except for [16] which describes an optimization sorting the most likely child node at the left side but uses a different data structure, we have not found anything similar to our optimizations, in particular related to the efficient application of SIMD vectorization to process deeper trees through partition- and node-level iterations that will be described in the next section. Sections V and VI will present different combinations of SIMD vectorization and multithreading to parallelize multiple tree traversals processing multiple input samples within a batch and/or multiple trees within an ensemble. Although some of the above related work applies similar parallelization concepts in a static fashion, we have not found a comparable approach involving runtime selection between the large number of combinations covered in this paper.

IV. Data structures for tree traversal

This section will introduce the optimized data structures that form the basic components of the presented inference function. These structures will be illustrated using an exemplary binary decision tree as shown in Fig. 2, which is comprised of seven split nodes (S_0 to S_6) represented by circles and eight leaf nodes (L_0 to L_7) represented by squares, having a maximum depth equal to three (the root node S_0 is considered to be at depth 0, nodes S_1 and S_2 at depth 1, and so on).

The processing of a decision tree for a given input sample, also denoted as tree traversal, starts at the root node and ends when a leaf node is reached. Each split node that is visited during a tree traversal, selects a feature of the input sample and compares it against a threshold value. Based on the outcome of the comparison, processing continues with either the left or the right child node. In this example, the left child node is selected if the feature value is less than or equal to the threshold, and the right child node is selected otherwise, as indicated by the symbols along the edges.

Each leaf node contains a label of a type that is determined by the model type. For example, for random forests the label can be an integer or floating-point value for regression, or a class identifier or set of class probabilities for hard or soft voting classification, respectively. The prediction result for a given input sample is derived from the labels in the leaf nodes that have been reached by traversing all trees in the ensemble.

A. Optimized breadth-first tree structure

A commonly used approach is to map the decision-tree nodes according to a *breadth-first* order within an array in the memory, starting with the root node being mapped at an offset 1. This allows the node interconnection structure to be encoded through the node offsets, without requiring explicit fields to define the child nodes of each split node (e.g., using pointers or offsets): the left and right child nodes of a given split node that is mapped at offset k, are mapped at offsets $2 \times k$ and $2 \times k + 1$. This approach, however, requires the decision tree to be a perfect tree (as described in Section III) such as the tree shown in Fig. 2. In this figure, the node offsets according to a breadth-first order are shown next to each node in red.

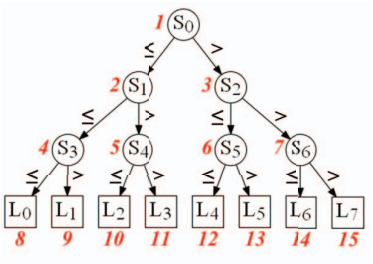

Fig. 2. Breadth-first decision tree.

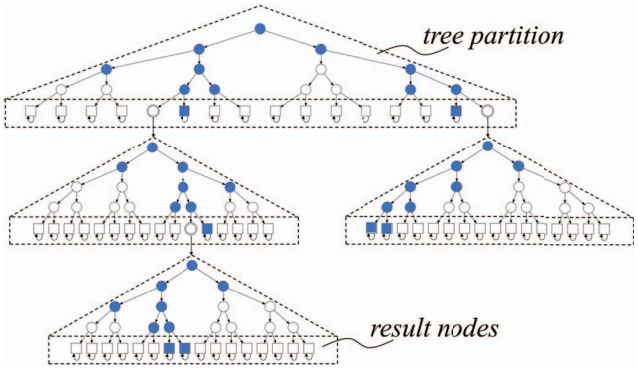

Fig. 3. Partitioned breadth-first decision tree.

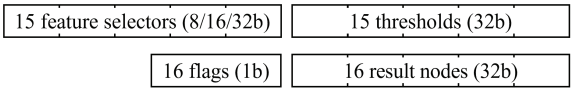

Fig. 4. Partition data structure.

The main advantages of this data structure are its compactness and the simple calculation of the offset of the child nodes from the parent node offset enabling efficient SIMD vectorization. An important disadvantage is the requirement of a perfect tree, which limits its application to relatively shallow trees. Another disadvantage is that the nodes on traversal paths between the root node and leaf nodes are typically not mapped near to each other, except in small trees, but are scattered throughout the memory array, resulting in minimal spatial locality and therefore low caching efficiency.

To overcome these disadvantages, an optimized breadth-first tree structure is proposed, henceforth referred to as *OBF*. The *OBF* scheme divides trees into multiple partitions, which are expanded to perfect subtrees as shown in the example in Fig. 3. In this example, the colored nodes comprise the original tree structure, and the other nodes were added to expand the shown partitions to perfect subtrees. The nodes at the lowest level in a partition are denoted as *result nodes*. A result node can either correspond to a leaf node in the original tree structure (and will then be represented by a square), or it can comprise a reference (e.g., pointer or offset) to another partition containing the next nodes from the original tree structure (in which case it will be represented by a circle). A tree traversal is completed if a result node of the former type is accessed and the label can be retrieved from that result node. If a result node of the latter type is accessed, then processing continues with the partition that it refers to.

As can be understood from the example in Fig. 3, this approach allows non-perfect trees to be handled more efficiently by only 'covering' the actual existing subtrees using partitions instead of expanding the entire tree to become a perfect tree for the maximum tree depth. As a result, however, tree traversals from the root to a leaf node can be of different lengths, resulting in certain tree traversals reaching leaf nodes earlier than other tree traversals, which makes SIMD vectorization more difficult. The *OBF* scheme handles this by iterating the processing of the last partition for the tree traversals that finished earlier at leaf nodes that are not at the maximum tree depth until all tree traversals are completed. These iterations are represented by the looping edges attached to result nodes corresponding to leaf nodes in Fig. 3.

The partition size is selected such that a partition fits in one or two cache lines to optimize spatial locality. In this paper, a fixed partition depth of 4 levels is used, corresponding to a perfect subtree with 15 split nodes and 16 result nodes. Fig. 4 illustrates the corresponding data structure for each partition, which consists of three arrays, with the first two arrays storing the feature selectors and threshold values (32 bit single-precision floats) of the 15 split nodes in the partition, and the third array comprising the 16 result nodes (32-bit values) as mentioned above. A vector containing 16 single-bit flags indicates for each result node if it corresponds to a leaf node and contains a label, or if it contains a reference to another partition. Using a separate array for the feature selectors makes it possible to reduce the memory footprint by adapting the data type to the actual number of features used in the given model, e.g., by using bytes for supporting models with up to 256 features, 16-bit shorts to support up to 64K features, and 32-bit words otherwise. The entries in each array are sorted according to the breadth-first order of the nodes and indexed using the offset calculation described above.

B. Optimized depth-first tree structure

Another common approach is to apply a *depth-first* ordering of the nodes within a memory array, starting with the root node being mapped at offset 0, which is illustrated in Fig. 5. This node interconnection structure is also encoded through the node offsets: the left child node of a split node mapped at offset k is mapped at offset $k+1$ and the right child node at offset $k+m_d$ with the 'offset increment' m_d being dependent on the depth d at which the parent node is located within the tree. For the example in Fig. 6, $m_0 = 8$, $m_1 = 4$, and $m_2 = 2$, with $m_{d+1} = \frac{m_d}{2}$. Similar to the *OBF* structure, this structure also requires a perfect tree to work correctly.

An optimized depth-first tree structure is proposed, which will hereby be denoted by *ODF*, which modifies the basic depth-first tree structure to exploit node-level access proba-

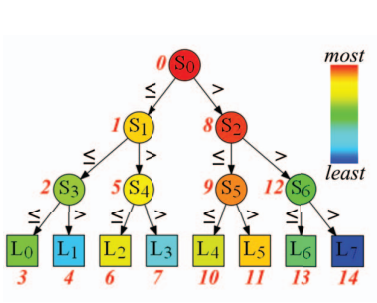

Fig. 5. Depth-first decision tree.

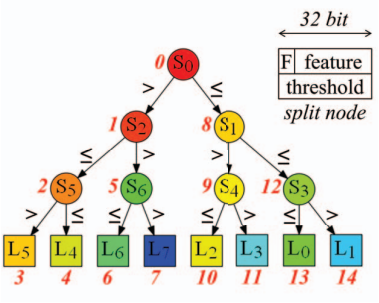

Fig. 6. 'Remapped' depth-first decision tree.

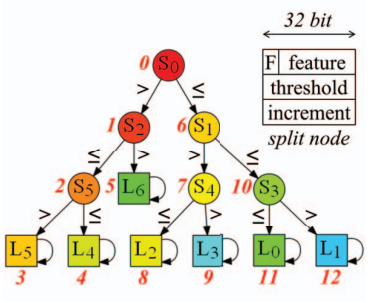

Fig. 7. Optimized depth-first decision tree.

bilities. In order to illustrate this, the nodes are colored to show a heatmap which reflects the node access probability as indicated by the legend shown in Fig. 5. This information can, for example, be derived from leaf cardinality information generated during training, which reflects the number of training samples that are 'mapped' on each path between the root node and any leaf node. Alternatively, node access probabilities can also be generated for the internal decision-tree ensemble representation to which an imported model is converted as was described in Section II, using training or similar data.

As can be seen in Fig. 5, paths that consist of sequences of 'left-child-nodes' (e.g., S_0, S_1, S_3, and L_0) are mapped on consecutive offsets which results in higher cache locality when traversing those paths. As also can be seen, these paths do not necessarily align with the most probable (frequently) travelled paths (e.g., S_0, S_2, S_5, and L_5) as indicated by the heatmap color. In order to 'correct' this alignment, the *ODF* scheme applies a per-node selection of the comparison operator (either less-than-or-equal-to or greater-than) such that the child node that is most likely to be accessed next corresponds to a positive comparison result and 'becomes' the left child node. The resulting structure for the example tree is shown in Fig. 6. In this example, the compare operators of nodes S_0, S_1, and S_5 have been changed as reflected by the symbols at the edges. Because of the above adaptation of the comparison operator in each node, now large parts of the most likely traversed paths are mapped on consecutive offsets, thus improving cache locality. To realize this, the applied comparison, $\leq$ or $>$, has to be encoded in the data structure. *ODF* does this by using the most significant bit of the feature selector field.

To support deeper trees, *ODF* removes the need for perfect trees by supporting an 'optional' selection of the offset increment for the right child node for each individual split node by storing it explicitly in the node structure in an additional field, while the offset of the left child node remains equal to the offset of the parent node incremented by one. This is illustrated by the example of a non-perfect tree in Fig. 7 in which the offset-increment selection is applied to all split nodes as can be seen from the differences between the offsets of the right child nodes and the parent nodes that can be selected independently.

This flexible offset-increment selection can be applied to all split nodes in the entire tree as shown in Fig. 7; however, it is more efficient to use the above discussed structure shown in Fig. 6 for traversing nodes near the root node and using the structure shown in Fig. 7 for nodes at the greater tree depths. The depth at which processing switches between the two structures constitutes an additional implementation parameter that is selected by the data-structure selector component in Fig. 1 when a model is imported. The first structure will be denoted as *ODF1* and the second as *ODF2*.

With the requirement for perfect trees removed, the traversal of different paths can now take a variable number of steps. In order to enable efficient SIMD vectorization, the leaf nodes are implemented and processed as a special kind of split nodes for which the comparison always produces a negative result. By using a zero offset increment when processing these nodes, the tree processing function will simply start looping over the same leaf node upon arrival. This continues until all parallel tree traversals have arrived at a leaf node. These loops are illustrated in Fig. 7. This concept is similar to the partition-level iterations applied by the *OBF* scheme discussed in Section IV-A, but applied at the node-level.

The *OBF* scheme maps the leaf nodes on the last consecutive offsets in each partition as can be seen in Fig. 2. The depth-first ordering applied by the *ODF* scheme, however, interleaves the offsets upon which the split nodes and leaf nodes are mapped as Figures 6 and 7 show. Because of this, an array-based data structure as shown in Fig. 4 for the *OBF* scheme does not make sense for the *ODF* structures discussed here. Instead, each node will be mapped on a single-node structure combining multiple fields storing the node's feature, node-level comparison type flag, threshold, and offset increment (for the *ODF2* structure), as is shown in Figures 6 and 7. For efficient processing, all fields are 32 bits wide.

V. SIMD Vectorization

The *OBF* and *ODF* schemes enable multiple tree traversals to be performed in parallel through the use of SIMD instructions operating on wide vector registers. These parallel tree traversals can process multiple trees within an ensemble for a single input sample, or multiple input samples using a single decision tree. A hybrid version, involving the parallel processing of multiple input samples using multiple decision trees is also feasible, but will not be considered here.

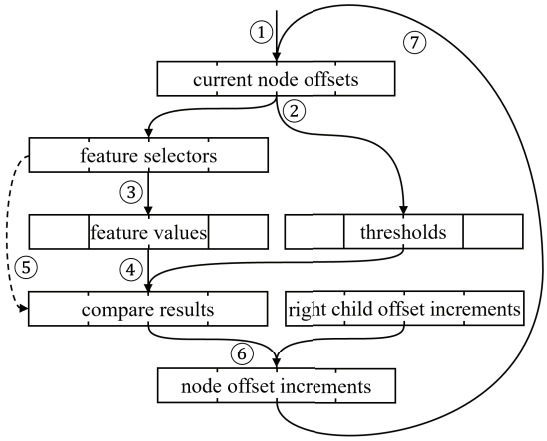

Fig. 8. SIMD vectorization for multiple tree traversals.

Fig. 8 illustrates a common set of steps and vectors for implementing SIMD-based traversal functions for the *OBF* and *ODF* tree structures:

① Load offset(s) of root node(s) into current-node-offsets vector.
② Load feature selectors and thresholds for the current nodes (gather instruction).
③ Extract feature values from current input sample(s) based on feature-selectors vector.
④ Compare feature-values and thresholds vectors.
⑤ Only for *ODF*, update the compare-results vector based on the node-level compare types defined by the flag bits that are extracted from the most-significant feature-selector bits (see Section IV-B).
⑥ Generate a node-offset-increments vector, that will be used to update the current-node-offsets vector, from the compare-results vector and a right-child-offset-increments vector.
⑦ Update current-node-offsets vector using the node-offset-increments vector. If the parallel tree traversal is completed, retrieve the results from the current node offsets in the data structure, otherwise iterate steps ② to ⑦.

In step ③, the feature values are extracted from the current input sample in case of parallelization over trees, and from successive input samples within the input batch in case of parallelization over input samples. The actual loops that will cover all trees in the ensemble and all input samples within a batch will be discussed in Section VI.

For the *OBF* scheme, the right-child-offset-increments vector comprises only elements equal to 1. These elements only need to be added to the current-node-offsets vector if the corresponding comparison result is negative. This can be implemented using masked add operations (e.g., AVX-512) or similar operations. This addition is executed as part of step ⑦, which first shifts the current-node-offsets vector to the left by one in order to multiply the node offsets by two as described in Section IV-A. If the parallel tree traversals through the *OBF* partitions all end in result nodes that correspond to leaf nodes, then the parallel tree traversals have completed. Otherwise step ⑦ involves retrieving the references to other partitions from result nodes that do not correspond to leaf nodes and start processing these partitions as described above, while iterating the processing of the same partitions for result nodes that correspond to leaf nodes, until all traversals have completed. Note that the latter partition-level processing is not covered in Fig. 8.

For *ODF1*, the right-child-offset-increments vector is loaded at the start of the prediction with an initial value that is determined by the tree depth and it is shifted to the right by one (divided by two) at the end of each loop as was discussed in Section IV-B. For *ODF2*, the right-child-offset-increments vector is retrieved in each loop from the node structures at the current-node-offset vector. *ODF* does not require a shift operation in step ⑦ as *OBF* does. Leaf nodes in the *ODF2* structure are processed as a special split node involving a zero offset increment which triggers repeated processing of these nodes as discussed in Section IV-B. Consequently, if the node-offset-increments vector only contains zero elements, then all parallel tree traversals have completed and the loop ends.

VI. Main loops and multithreading

During inference, all trees within an ensemble have to be traversed for each sample in a given input batch. Consequently, there exist two basic loops in a predict function, one iterating through the samples in an input batch and one iterating through the trees in the ensemble:

for $i \in \{0, \ldots, I-1\}$ **do**
  for $t \in \{0, \ldots, T-1\}$ **do**
    tree_traversal(t, i)
  end for
end for

with I being the input batch size, T being the number of trees in the ensemble, i being the index of the current input sample, and t being the index of the current tree being processed. Alternatively, the order of the loops can also be exchanged:

for $t \in \{0, \ldots, T-1\}$ **do**
  for $i \in \{0, \ldots, I-1\}$ **do**
    tree_traversal(t, i)
  end for
end for

The latter code fragment will involve higher spatial and temporal locality properties in relation to the trees that are being processed in the outer loop because each tree structure will be processed during a longer continuous period of time, whereas this applies in a similar way to the input samples for the former code fragment. Although the final prediction result will be the same in both cases, the two loop orders can result in different prediction speeds on a given computer system.

If the predict function uses SIMD instructions to process multiple trees in parallel or multiple samples within an input batch as was discussed in the previous section, then the loop variable t of the 'tree loop' respectively i of the 'input loop' will be incremented by the number of tree traversals that are processed in parallel in each loop iteration using SIMD instructions, denoted here by p. In case the number of trees within an ensemble would not be a multiple of p then this can be resolved by 'padding' the ensemble with additional trees that do not impact the final prediction results (e.g., zero leaf labels). Alternatively, the 'remaining' trees can be processed by non-simd-vectorized predict functions. The same principle can be applied to the input batch in case of SIMD parallelization over the input samples.

In addition to SIMD vectorization, the predict function can also be accelerated using multithreading. As part of the experiments that are presented in the next section, one of the two loops in the above code fragments will be parallelized using OpenMP [38]. If the tree loop is parallelized, special precautions have to be taken to prevent race conditions to occur when the results of two parallel tree traversals that process the same input sample are added to the (intermediate) prediction results of that sample. These precautions (e.g., an OpenMP reduction clause) can, however, negatively affect the performance gain that is achieved through multithreading.

Considering the two possible combinations of the tree and sample loops as inner and outer loops, the three possible applications of OpenMP to these loops (outer loop, inner loop, no application), and the three possible applications of SIMD vectorization (vectorized processing of multiple trees for one input sample, vectorized processing of multiple input samples for a single tree, no application), then this results in a total of $2 \times 3 \times 3 = 18$ possible combinations. The loop order will now be represented by a two character combination **ti** or **it** for an outer tree/inner input sample loop respectively outer input sample/inner tree loop. If a loop is parallelized using OpenMP, then the corresponding character will be written as uppercase. If SIMD vectorization is applied to parallelize the processing of input samples or trees then the corresponding character will be underlined. For example, a predict function involving a 'tree outer loop' and 'sample inner loop' with the latter being parallelized using OpenMP, also using SIMD vectorization to simultaneously process multiple trees for a single input sample is represented by the character combination **t<u>I</u>**.

TABLE I
DATASETS USED FOR EVALUATION

dataset	type	#class	#features	#samples	
				training	test
epsilon [39]	class.	2	2000	300 K	100 K
HIGGS [40]	class.	2	28	8250 K	2750 K
SUSY [41]	class.	2	18	3750 K	1250 K
Covertype [42]	class.	7	54	11.3 K	3.8 K
YearPredictionMSD [43]	regr.	-	90	463 K	51 K

VII. EXPERIMENTS

A. Setup

A large number of experiments were performed using the publicly available classification and regression datasets listed in Table I that are frequently used in publications for benchmarking inference algorithms. Multiple gradient boosting and random forest models with various tree count and depth combinations were trained for each dataset using XGBoost 1.7.4, LightGBM 3.3.5, and Scikit-Learn 1.2.2. The trained models were both serialized using pickle and exported to ONNX format. The resulting files were then loaded for performing inference experiments with the native runtime (i.e., the scheme that was used to train the model), with ONNX Runtime 1.14.1 and with our inference function. The LightGBM models were also scored using lleaves 1.0.0.

The experiments were carried out on a single node of a shared-memory NUMA system, having 128GB of main memory and two Intel® Xeon® Gold 6230 CPUs running at 2.10GHz. Each CPU had 20 cores, dedicated 32KB L1 and 1MB L2 caches and a shared 27.5MB L3 cache, and supported AVX2 and AVX-512 SIMD instructions. The system was running Ubuntu 20.04.6 LTS. As discussed in Section II, the inference function was implemented as a C++ extension module that is accessed through a Python script. The C++ code was compiled using gcc 9.4.0 with the -O3 flag. All timing measurements were performed as part of the Python script. For each model many prediction requests were executed such that the test data available for the corresponding data sets as listed in Table I was covered at least once by all batches and such that there were at least 64 prediction requests for each batch size. The batches were selected in various ways from the test data, but the exact order and alignment did not appear to have much impact on the measured performance. Finally, the arithmetic mean was determined over all measured predict times for each given batch size.

Inference is typically executed as part of an application framework, implying that in addition to the predict function other tasks are executed on the CPU, which also consume computer resources and therefore could degrade predict performance. It is assumed that there are mechanisms in place (e.g., by setting processor affinity attributes) that dedicate one or multiple cores to the inference task and that the execution of the application framework will not significantly impact the predict operation nor affect decision-tree data stored in the caches of those cores. This seems to be in line with the way in which experimental results are obtained and presented by most of the published related work.

B. Dynamic predict function selection

Fig. 9 shows the average predict time (latency) per input sample that was measured on the test system for four single-threaded predict functions implemented as part of the *ODF* scheme that involve various ways of parallelization using AVX2 SIMD instructions as described in Section VI, for a

range of batch sizes. Each function is identified using the representation of the inner/outer loop, SIMD vectorization and multithreading defined in that same section. The model was trained using XGBoost for the epsilon dataset with the number of trees, T, tree depth, d, and number of CPU threads, thr, listed in Fig. 9. Fig. 10 shows a similar graph for 12 multithreaded predict functions (also for *ODF* and implemented using AVX2 instructions) for a model trained for the HIGGS dataset. Both figures illustrate that there can be considerable differences in the performance of the predict functions for the same batch sizes.

Intuitively it can be expected that for short batch sizes, which provide less options to parallelize over the input samples, predict functions that parallelize traversals over multiple trees using either SIMD vectorization and/or multithreading will perform better. This can be seen clearly in Fig. 9 for batch sizes below 8 samples, for which predict functions **ti** and **it**, which use SIMD vectorization to process multiple trees in parallel, both perform well. It is interesting to see in Fig. 10, that when in addition also multithreading is enabled over the tree loop (functions **Ti** and **iT**), the performance becomes worse in comparison to **ti** and **it**. The latter can be explained by realizing that the multithreading over the tree loop will introduce additional tree structures into the cache, which can result in cache conflicts ('thrashing') between the parallel tree traversals which consequently degrades performance. The extent to which these cache conflicts occur between parallel tree traversals depends on several factors, including obviously the number of parallel traversals, the size of each tree and the number of active paths (e.g., the heatmap in Fig. 5). In a similar way, when tree traversals are parallelized to process multiple input samples, then the size of these samples (i.e., the number of features) and how these samples are accessed can also result in cache conflicts and affect performance. The selection of the best predict function taking into account these effects, is automatically done based on benchmarking data as was described in Section II.

C. Comparison with other schemes

In this section the performance of the inference function that is proposed in this paper and comprises the various components discussed in the previous sections is compared to state-of-the-art algorithms. Figures 11 to 13 show the measured performance expressed as average predict latency per input sample for three models trained and scored using XGBoost, LightGBM and Scikit-Learn (random forest) as described in Section VII-A. The graphs also show the inference performance for the ONNX Runtime library and for the dynamically selected *OBF* and *ODF* predict functions implemented using AVX2 and AVX-512 SIMD instructions.

The *OBF* and *ODF* schemes outperform XGBoost, LightGBM, Scikit-Learn, and ONNX Runtime throughout the entire range of batch sizes, with the largest performance gain occurring for short batch sizes. The latter was to be expected, given that most decision-tree algorithms are optimized for processing

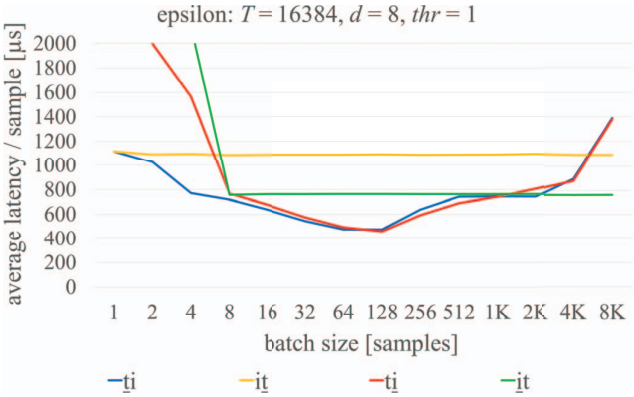

Fig. 9. Single-threaded predict functions.

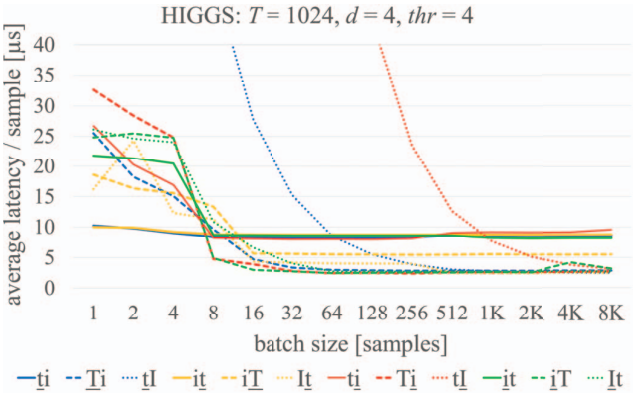

Fig. 10. Multithreaded predict functions.

larger batch sizes. The fact that *OBF* and *ODF* also achieve the best performance for longer batch sizes, is clearly due to the dynamic predict function selection, which also appears to be very effective in this case.

Table II shows performance results, in terms of latency per sample, for a large variety of model parameters, batch sizes and number of threads. This table consists of five horizontal sections, one for each dataset listed in Table I, and three vertical sections corresponding to XGBoost, LightGBM and Scikit-Learn, respectively. Each intersection of a horizontal and vertical table section corresponds to a model that is trained using one of the three schemes for one of the five datasets. The top row in each intersection lists the number of trees, T, and the maximum tree depth, d, of the model, as well as the number of CPU threads, thr, used for prediction. It also includes an indication if the trees in the model are *perfect* ($\#leafs = 2^d$, $\bar{d}_{leaf} = d$), *sparse* ($\#leafs \ll 2^d$, $\bar{d}_{leaf} \ll d$) or *dense* (remaining trees), with $\#leafs$ being the average number of leaf nodes in each tree and $\bar{d}_{leaf}$ being the average depth at which these leaf nodes reside.

The next four rows are divided into multiple columns,

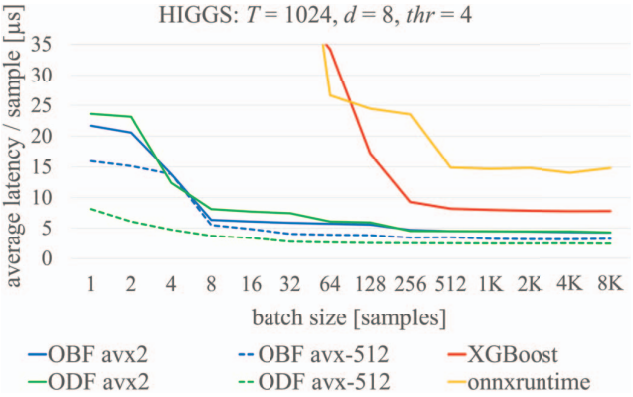

Fig. 11. Comparison with XGBoost.

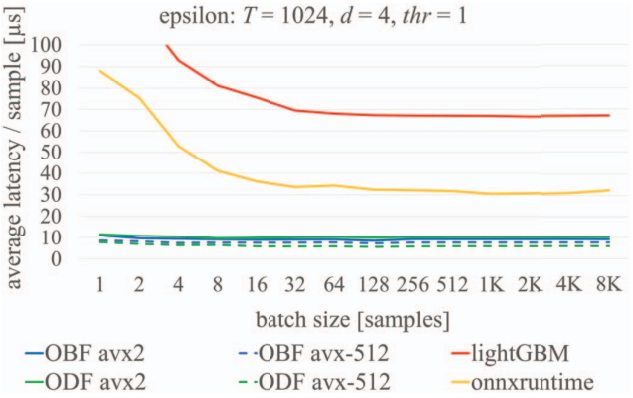

Fig. 12. Comparison with LightGBM.

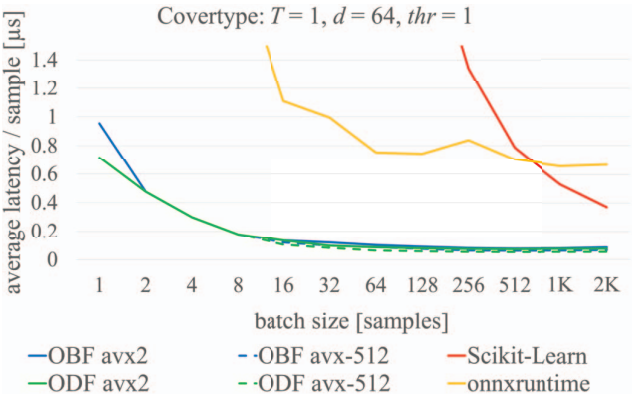

Fig. 13. Comparison with Scikit-Learn.

which show the measured average prediction times per sample for the native runtime, for the ONNX Runtime library, for lleaves (LightGBM models only), and for *OBF* and *ODF*, respectively, for four different batch sizes (the maximum batch size for scoring models trained using the Covertype dataset was reduced because of the limited number of test samples - see Table I). The last row shows the model sizes in bytes, which equal the serialized (pickled) file sizes, except for the ONNX Runtime library which does not support serialization and for which the ONNX file size is used instead, and except for lleaves for which the size of the exported binary ELF file is used. The best (i.e., smallest) latency and model size values are highlighted using a bold font.

Because random forests can involve deeper trees than gradient boosting models [16], the latter models (XGBoost, LightGBM) were trained for a larger number of shallower trees and the former (Scikit-Learn) for fewer but deeper trees. It also appeared that for several experiments involving multithreading, Scikit-Learn and lleaves performed better using single-threaded operation. Although these two schemes do not support an automatic reduction of the number of actually used threads to improve performance, we decided to list the better single-threaded latency values (marked with an '*') for both schemes in Table II for having a fairer comparison. For the same reason, we also limited the number of threads used for scoring the Scikit-Learn models.

As can be seen from Table II, the dynamically selected *OBF* and *ODF* predict functions outperform all other schemes across all models. Performance gains of one to two orders of magnitude are achieved for single-sample prediction compared to XGBoost, LightGBM and Scikit-Learn, and a factor two to ten for longer batches for most models. The AVX2 implementations of *OBF* and *ODF* predict functions outperform ONNX Runtime on average by more than a factor four and lleaves by about a factor three for the models in Table II. The AVX-512 implementations of the *OBF* and *ODF* predict functions outperform the AVX2 implementations on average by about a factor 1.5. Table II also shows that for all models, *OBF* or *ODF* achieve the smallest (serialized) model sizes.

Detailed analysis and further experiments have revealed that compared to a non-parallel implementation of the *OBF* and *ODF* predict functions, the performance was improved by approximately a factor 2 and 3 by only using AVX2 or AVX-512 SIMD vectorization, respectively, by about a factor 6 when only using multithreading, and by a factor of 11.4 and 14.2 when using both multithreading and AVX2 or AVX-512 SIMD vectorization, respectively. Note that these are average values for all experiments listed in Table II. These numbers can change substantially for individual models, batch sizes and number of CPU threads.

For determining the average latency numbers listed in Table II from the measured prediction latencies, the following coefficients of variation (standard deviation divided by mean) were observed for the respective batch sizes 1, 128, 1024 and 8192 (2048 for the Covertype models): 1.2, 0.20, 0.071, and 0.027. Because delays can be amortized more easily over longer batches, obviously the shortest batch sizes show the most variability. This is also reflected in the observed 95th and 99th percentiles which were on average 28%, 20%, 9.4% and 4.5%, respectively 67%, 59%, 13%, and 4.8% greater than the mean value.

TABLE II
EXPERIMENTAL RESULTS

	batch size	XGBoost: XG Boost	XGBoost: ONNX Runt.	XGBoost: *OBF* avx2/512	XGBoost: *ODF* avx2/512	LightGBM: Light GBM	LightGBM: llea ves	LightGBM: ONNX Runt.	LightGBM: *OBF* avx2/512	LightGBM: *ODF* avx2/512	Scikit-Learn: Scikit Learn	Scikit-Learn: ONNX Runt.	Scikit-Learn: *OBF* avx2/512	Scikit-Learn: *ODF* avx2/512
epsilon		$T = 1024$, $d = 10$, $thr = 4$ dense ($\#leafs = 985$, $\bar{d}_{leaf} = 9.97$)				$T = 1024$, $d = 8$, $thr = 1$ sparse ($\#leafs = 31.0$, $\bar{d}_{leaf} = 5.34$)					$T = 128$, $d = 12$, $thr = 1$ dense ($\#leafs = 3.01K$, $\bar{d}_{leaf} = 11.8$)			
average	1	484	162	20 / 15	**17 / 12**	203	93	182	36 / 26	**31 / 23**	8397	71	**8.8 / 6.4**	9.3 / 6.7
latency /	128	62	63	17 / 12	**14 / 8.2**	110	27	47	24 / 17	**22 / 13**	127	37	**7.6 / 5.7**	8.4 / 5.9
sample	1024	36	63	17 / 12	**14 / 8.1**	109	28	46	24 / 17	**22 / 12**	58	36	**7.6 / 5.5**	8.4 / 5.6
[μs]	8192	33	63	17 / 12	**14 / 8.1**	110	35	47	24 / 17	**22 / 13**	44	36	**7.6 / 5.5**	8.4 / 5.6
size [B]		66M	68M	43M	**16M**	3.7M	3.1M	2.3M	1.5M	**0.56M**	53M	29M	**5.2M**	6.0M
HIGGS		$T = 16384$, $d = 8$, $thr = 16$ dense ($\#leafs = 253$, $\bar{d}_{leaf} = 8.0$)				$T = 1024$, $d = 4$, $thr = 4$ perfect ($\#leafs = 16$, $\bar{d}_{leaf} = 4.0$)					$T = 128$, $d = 16$, $thr = 8$ dense ($\#leafs = 31.9K$, $\bar{d}_{leaf} = 15.6$)			
average	1	4379	668	**36 / 27**	43 / 29	140	39*	26	**5.4** / 4.7	6.0 / **4.3**	8537*	27	**10** / 9.8	**10 / 8.6**
latency /	128	429	104	**24** / 17	25 / **14**	15	8.5*	7.1	**2.3** / 1.9	2.4 / **1.4**	134*	9.4	**1.4 / 0.95**	1.6 / 0.96
sample	1024	54	103	**23** / 16	**23 / 12**	14	7.6*	6.6	**2.1** / 1.6	2.3 / **1.2**	64.6	8.0	**1.3 / 0.87**	1.5 / 0.91
[μs]	8192	44	103	**23** / 16	**23 / 12**	14	4.7	6.6	**2.1** / 1.6	2.3 / **1.2**	9.1	8.0	**1.1** / 0.81	**1.1 / 0.76**
size [B]		279M	271M	**27M**	48M	1.9M	1.4M	1.1M	**0.14M**	0.18M	561M	319M	**57M**	62M
SUSY		$T = 8192$, $d = 7$, $thr = 8$ dense ($\#leafs = 125$, $\bar{d}_{leaf} = 7.0$)				$T = 128$, $d = 12$, $thr = 2$ sparse ($\#leafs = 31.0$, $\bar{d}_{leaf} = 7.0$)					$T = 16$, $d = 48$, $thr = 4$ sparse ($\#leafs = 419K$, $\bar{d}_{leaf} = 28.1$)			
average	1	1429	425	27 / 21	**25 / 16**	88	12*	19	**3.6** / 3.5	4.0 / **2.8**	1327*	33	5.2 / 5.2	**4.5 / 4.1**
latency /	128	139	59	18 / 13	**15 / 8.6**	5.5	2.0*	3.0	**1.6** / 1.2	**1.6 / 0.91**	29*	4.8	1.3 / 0.96	**1.0 / 0.73**
sample	1024	33	59	18 / 12	**15 / 8.5**	5.2	1.7*	2.7	**1.6** / 1.1	**1.6 / 0.88**	7.0	4.6	1.2 / 0.87	**0.91 / 0.63**
[μs]	8192	32	59	18 / 12	**15 / 8.4**	4.7	1.7*	2.7	**1.6** / 1.1	**1.6 / 0.87**	3.8	4.6	1.1 / 0.84	**0.86 / 0.61**
size [B]		71M	66M	19M	**12M**	457K	376K	279K	137K	**83K**	920M	530M	229M	**102M**
Covertype		$T = 7000$, $d = 12$, $thr = 2$ sparse ($\#leafs = 16.4$, $\bar{d}_{leaf} = 6.7$)				$T = 700$, $d = 8$, $thr = 16$ sparse ($\#leafs = 31.0$, $\bar{d}_{leaf} = 6.4$)					$T = 10$, $d = 32$, $thr = 1$ sparse ($\#leafs = 2.07K$, $\bar{d}_{leaf} = 16.6$)			
average	1	1420	367	**121 / 90**	129 / 100	238	26*	31	**22** / 17	**22 / 15**	843	12	1.9 / 1.9	**1.6 / 1.7**
latency /	128	94	116	**58** / 41	65 / **34**	5.3	5.7*	4.5	**1.6** / 1.2	**1.6 / 1.1**	12	2.4	0.86 / 0.62	**0.71 / 0.44**
sample	1024	90	113	**57** / 41	63 / **34**	4.5	4.6*	4.1	**1.1** / 0.8	**1.1 / 0.7**	3.3	2.1	0.78 / 0.54	**0.67 / 0.41**
[μs]	2048	92	116	**56** / 43	62 / **33**	4.3	4.2	4.1	**1.1** / 0.8	**1.1 / 0.7**	2.9	2.1	0.77 / 0.53	**0.67 / 0.40**
size [B]		12M	7M	**3M**	**3M**	2.2M	1.8M	1.5M	0.5M	**0.4M**	4.4M	2.9M	1.2M	**0.8M**
YearPredictionMSD		$T = 512$, $d = 9$, $thr = 20$ dense ($\#leafs = 451$, $\bar{d}_{leaf} = 8.9$)				$T = 2000$, $d = 3$, $thr = 12$ perfect ($\#leafs = 8.0$, $\bar{d}_{leaf} = 3.0$)					$T = 256$, $d = 16$, $thr = 1$ dense ($\#leafs = 21.2K$, $\bar{d}_{leaf} = 15.3$)			
average	1	323	28	18 / 15	**14 / 8.8**	212	39*	33	**11** / 10	**11 / 6.9**	13370	201	**27** / 20	28 / **19**
latency /	128	14	4.0	1.7 / 1.2	**1.4 / 0.9**	11	12*	7.5	2.0 / 1.5	**1.7 / 1.0**	237	126	**22 / 15**	23 / **15**
sample	1024	3.3	3.1	1.5 / 1.0	**1.2 / 0.7**	9.8	8.5	5.0	1.8 / 1.4	**1.6 / 0.90**	117	125	**18 / 13**	19 / **13**
[μs]	8192	1.7	2.8	1.1 / 0.8	**1.0 / 0.6**	9.8	3.5	3.9	1.8 / 1.4	**1.4 / 0.80**	60	124	**16** / 12	**16 / 11**
size [B]		15M	15M	16M	**3.5M**	1.7M	1.3M	1.1M	0.28M	**0.18M**	662M	427M	89M	**83M**

T : number of trees, d : tree depth, thr : number of CPU threads, $\#leafs$: avg. number of leaf nodes per tree, $\bar{d}_{leaf}$: avg. leaf node depth
* : measured latency for single-threaded operation used instead of higher measured latency for multithreaded operation using thr threads

VIII. CONCLUSIONS

In this paper, we have presented the *OBF* and *ODF* tree structures that combine the advantages of traditional breadth-first and depth-first trees, namely compactness and ease of processing, with the support for deeper trees by removing the need for the decision trees to be perfect trees that have the disadvantage of growing exponentially with the tree depth. The two schemes can be accelerated efficiently using SIMD vectorization, applying partition-level and node-level iterations, respectively, to handle parallel traversals that do not complete at the same time. Furthermore, the *ODF* scheme includes a mechanism for improving the spatial locality properties of the data structure by exploiting node-level access probabilities.

The *OBF* and *ODF* schemes are implemented as part of a decision tree inference function which can import decision tree models that are trained using different frameworks and exported in formats such as PMML or ONNX. At model import time, the inference function selects which of the two schemes is used and determines the corresponding parameters.

A key innovation of the inference function is the dynamic selection of the predict function at predict time from a predetermined collection, each applying a different combination of SIMD vectorization and multithreading to perform multiple tree traversals in parallel. This selection is based on model and compute platform parameters as well as predict-time parameters such as input batch size and the number of CPU threads, and is guided by benchmarking data.

An extensive number of experiments have shown that this concept achieves substantial performance gains over state-of-the-art decision-tree inference algorithms, while it can efficiently handle a wide range of model parameters and application requirements, including those of emerging applications requiring real-time predictions for individual samples. Additionally, the *OBF* and *ODF* structures realize a sizeable decrease in model size, which can be a key feature for cases where many models need to be scored for multiple users in parallel, for example, in a cloud environment.

REFERENCES

[1] J. Morgan and J. Sonquist, "Problems in the analysis of survey data, and a proposal," *Journal of the American Statistical Association*, vol. 58, pp. 415–434, 1963.

[2] R. C. Messenger and L. Mandell, "A modal search technique for predictive nominal scale multivariate analysis," *Journal of the American Statistical Association*, vol. 67, pp. 768–772, 1972.

[3] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[4] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[5] ——, "Arcing the edge," Statistics Department, University of California at Berkeley, Tech. Rep. 486, 1997.

[6] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics and Data Analysis*, vol. 38, pp. 367–378, 1999.

[7] ——, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, vol. 29, pp. 1189–1232, 2001.

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

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

[10] Kaggle. (2021) State of data science and machine learning 2021. [Online]. Available: https://www.kaggle.com/kaggle-survey-2021

[11] 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*, 2021, pp. 426–440.

[12] M. Owaida, H. Zhang, C. Zhang, and G. Alonso, "Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms," in *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

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

[14] H. Cho and M. Li, "Treelite: Toolbox for decision tree deployment," in *SysML*, 2018. [Online]. Available: https://www.amazon.science/publications/treelite-toolbox-for-decision-tree-deployment

[15] K.-H. Chen, C. Su, C. Hakert, S. Buschjäger, C.-L. Lee, J.-K. Lee, K. Morik, and J.-J. Chen, "Efficient realization of decision trees for real-time inference," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 6, 2022.

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

[17] F. Lettich, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Parallel traversal of large ensembles of decision trees," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2075–2089, 2019.

[18] C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, F. Silvestri, and S. Trani, "Post-learning optimization of tree ensembles for efficient ranking," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2016, pp. 949–952.

[19] T. Ye, H. Zhou, W. Y. Zou, B. Gao, and R. Zhang, "RapidScorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2018, pp. 941–950.

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

[21] Scikit-learn, machine learning in Python. [Online]. Available: https://scikit-learn.org

[22] XGBoost, scalable and flexible gradient boosting. [Online]. Available: https://xgboost.ai

[23] LightGBM. [Online]. Available: https://www.microsoft.com/en-us/research/project/lightgbm

[24] Predictive model markup language (PMML). [Online]. Available: https://dmg.org/pmml

[25] Open neural network exchange (ONNX). [Online]. Available: https://onnx.ai

[26] Sparse Forests with FIL. [Online]. Available: https://developer.nvidia.com/blog/sparse-forests-with-fil

[27] G. Pedretti, C. E. Graves, S. Serebryakov, R. Mao, X. Sheng, M. Foltin, C. Li, and J. P. Strachan, "Tree-based machine learning performed in-memory with memristive analog CAM," *Nature Communications*, vol. 12, no. 1, 2021.

[28] 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.

[29] 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 and Development in Information Retrieval*, 2014, pp. 1123–1126.

[30] lleaves GitHub repository. [Online]. Available: https://github.com/siboehm/lleaves

[31] ONNX Runtime: Optimize and accelerate machine learning inferencing and training. [Online]. Available: https://onnxruntime.ai

[32] 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*, 2015, pp. 73–82.

[33] ——, "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*, 2016, pp. 833–836.

[34] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt, "Early exit optimizations for additive machine learned ranking systems," in *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, 2010, pp. 411–420.

[35] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: Unbiased boosting with categorical features," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 6639–6649.

[36] P. Langley and S. Sage, "Oblivious decision trees and abstract cases," in *Working Notes of the AAAI-94 Workshop on Case-Based Reasoning*.

[37] Y. Lou and M. Obukhov, "BDT: Gradient boosted decision tables for high accuracy and scoring efficiency," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1893–1901.

[38] The OpenMP API specification for parallel programming. [Online]. Available: https://www.openmp.org

[39] Epsilon dataset (PASCAL challenge 2008). [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html

[40] D. Whiteson, "HIGGS," UCI Machine Learning Repository, 2014. [Online]. Available: https://archive.ics.uci.edu/dataset/280/higgs

[41] ——, "SUSY," UCI Machine Learning Repository, 2014. [Online]. Available: https://archive.ics.uci.edu/dataset/279/susy

[42] J. Blackard, "Covertype," UCI Machine Learning Repository, 1998. [Online]. Available: https://archive.ics.uci.edu/dataset/31/covertype

[43] T. Bertin-Mahieux, "YearPredictionMSD," UCI Machine Learning Repository, 2011. [Online]. Available: https://archive.ics.uci.edu/dataset/203/yearpredictionmsd