2026 Design, Automation & Test in Europe Conference (DATE 2026)

Shakan: Training-Inference co-design for Oblique Random Forests on Embedded Devices

Alessandro Annechini*, Alessandro Verosimile*, Marco D. Santambrogio
Department of Electronics, Information and Bioengineering - DEIB
Politecnico di Milano, Milano, Italy
Email: {alessandro.annechini, alessandro.verosimile, marco.santambrogio}@polimi.it

***Abstract*—Embedded systems are increasingly leveraging Artificial Intelligence of Things (AIoT) to enable real-time decision-making in critical applications, such as autonomous navigation and medical diagnostics. In these contexts, Random Forests (RFs) have been widely adopted due to their inherent parallelism. However, RFs rely on axis-aligned splits, which limit their ability to model complex decision boundaries. Oblique Random Forests (ORFs), which employ hyperplane-based splits, offer a more expressive alternative by improving classification accuracy. Despite their advantages, inference of ORFs is resource-consuming, prohibiting the implementation of such models on resource-constrained hardware devices.**

In this work, we present Shakan, a novel framework for Oblique Decision Trees (ODTs) inference on embedded systems. We introduce a new training technique designed to mitigate both training complexity and overfitting while enabling low-latency inference in hardware, along with a new architecture that maximizes performance and optimizes resource usage. Shakan enables, on resource-constrained devices, the inference of several ORFs configurations that can provide either a significant increase in accuracy or a notable speedup in terms of inference latency compared to state-of-the-art accelerators for traditional RFs on embedded devices. The most accurate configurations provide average accuracy improvements above 5% with similar latency, while the fastest configurations achieve speedups of $1140\times$, $214\times$, and $29\times$ for tree depths of 5, 7, and 9, respectively, with comparable accuracy.

***Index Terms*—Oblique Random Forests, Artificial Intelligence of Things, HW-SW co-design, Embedded Devices**

I. Introduction

As Machine Learning (ML) becomes more common in everyday technology, the need to run smart models directly on embedded and edge devices continues to grow. These devices include small, low-power systems such as wearable health monitors, smart sensors, home automation devices, and mobile robots. In many of these applications, it is essential for the device to make decisions quickly and locally, without relying on an internet connection. Cloud-based AI solutions, while powerful, often introduce delays due to communication overhead or may be unavailable due to connectivity issues. In time-critical situations, such as detecting a fall, avoiding an obstacle, or reacting to sensor readings, immediate, on-device decision-making is not just preferred, but necessary [1].

When implemented on embedded devices, ML algorithms must operate under strict constraints on memory, energy, and processing power. Decision Trees (DTs) and their ensemble variants, particularly Random Forests (RFs), are attractive in this context due to their simplicity and inherent parallelism [2]. However, conventional RFs rely on axis-aligned splits, which can limit their expressiveness.

Oblique Random Forests (ORFs) address this limitation by allowing decision boundaries defined by linear combinations of features at each node, enabling more compact DTs and potentially better generalization [3]. While oblique splits have been shown to improve accuracy and reduce tree depth, the need for complex operations during inference and their computational cost during training make standard implementations of ORFs impractical in embedded environments.

In this work, we introduce Shakan, an implementation of ORFs specifically optimized for embedded devices. We present two main contributions:

- A **novel training algorithm** for ORFs that balances model expressiveness with computational efficiency, allowing for **low-latency hardware inference** while reducing overfitting (Section III-A);
- A new Field-Programmable Gate Array (FPGA)-tested architecture that maximizes **memory efficiency** while exploiting hardware-level parallelism, allocating Oblique Decision Trees (ODTs) on parallel pipelines of Processing Elements (PEs) (Section III-B).

We evaluate our approach on classification tasks relevant to embedded applications. Our results prove that Shakan can be successfully implemented on **resource-constrained systems**, offering **improved accuracy** and **reduced latency** over axis-aligned RFs, without exceeding hardware resource limits.

II. Background and Related works

DTs for classification are supervised learning models that predict class labels by recursively partitioning the input space. In most practical implementations, DTs have a binary tree structure where each internal node applies a threshold-based condition to a single numerical feature, dividing the data into two subsets. The process continues until a stopping condition is met, such as a maximum depth or pure leaves (i.e. leaves reached only by training samples assigned to the same class). Each leaf node assigns a class label based on the majority class of the training samples that reach it. RFs are an ensemble of multiple DTs: by aggregating the predictions of several DTs,

*Both authors equally contributed to this work.

978-3-9826741-1-7/DATE26/© 2026 EDAA

generated from different subsets of training data, RFs reduce overfitting and improve generalization.

Several works propose hardware implementation of traditional DTs ensembles, employing either logic-centric or memory-centric approaches. In logic-centric architectures, the DTs are directly mapped as logic into the hardware, while in memory-centric architectures the nodes of each DT are encoded and stored in memory arrays, while separate PEs are responsible for inference execution. The former approach allows for higher throughput, but is it is inefficient in terms of resource consumption, limiting the ensemble sizes on small devices [4]. Solutions based on partial dynamic reconfiguration of FPGAs exist [5], but cannot be applied to other categories of devices. On the other hand, the memory-centric approach allows for the execution of large ensembles on small devices. Some memory-centric architecture exploit horizontal parallelism [6], [7], while others focus on vertical parallelism [8], [9]. Horizontal parallelism refers to the execution of multiple DTs in parallel, while vertical parallelism exploits the fact that the computation at each level of a DT depends only on the previous level's decision, allowing for efficient operations pipelining. Moyogi [10], that exploits both horizontal and vertical parallelisms, represents at the moment the state-of-the-art solution in terms of memory-centric architecture for RF inference on embedded devices. Moyogi employs Multi Depth Random Forests (MD-RFs) introduced in [8] to improve memory utilization, along with multiple parallel pipelines of PEs to accelerate DT evaluations, providing state-of-the-art latency and accuracy in inference tasks. MD-RF is an ML model similar to traditional RFs, where the DTs have different depths in order to fill most of the available memory in the Memory Elements (MEs). Figure 1 shows the basic block of Moyogi, where a set of DTs is evaluated by a series of pipelined PEs. Each PE and ME accommodates one layer of each DT, where the nodes are encoded as 64-bit instructions in the ME. Multiple basic blocks are joined both in series and in parallel, minimizing the latency overhead. To account for the loss in accuracy due to the reduced depths, a Neural Network is employed during training to weight the votes of DTs depending on their maximum depth.

In addition to traditional DTs, a distinct class known as ODTs has been proposed. ODTs differ from traditional DTs in how they split data at each node. While DTs use so called *axis-aligned splits*, comparing a single feature against a threshold, ODTs employ *hyperplane-based splits*, where a linear combination of the F features of each sample X is compared against a threshold $\texttt{th}$:

$$\sum_{i=1}^{F} w_i \cdot X_i < \texttt{th} \tag{1}$$

The need for efficient linear combinations of real numbers requires hardware resources that are often inaccessible on embedded devices, restricting the implementation of this ML model to single ODTs instead of ORFs. In [11], the authors introduce a memory-centric technique to implement single ODTs with concatenated *universal nodes*, and in [12] an optimization based on tree sparsification is discussed. In [13], the authors propose an implementation of ODTs more efficient in terms of hardware resources, with latencies ranging from $50\mu s$ to $3000\mu s$ depending on the dataset. The proposed architecture, however, applies quantization on the input features and only analyses trees with a maximum depth of 3. Overall, none of the proposed solutions prove to be advantageous in terms of latency or accuracy when compared to traditional RF hardware implementations.

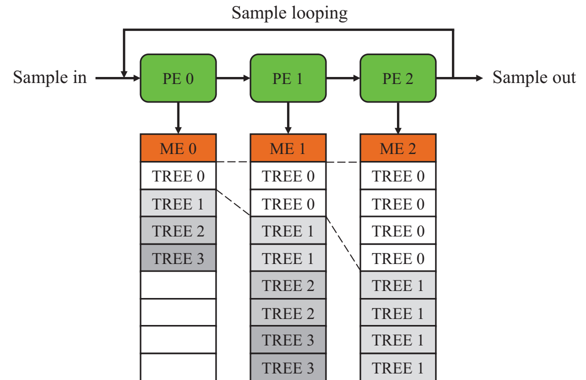

Fig. 1. Moyogi basic-block architecture, implementing a MD-RF classifier with two DTs at depth 3 (TREE 0 and TREE 1) and two DTs at depth 2 (TREE 2 and TREE 3).

Additionally, training an ODT, and therefore ORFs, is a challenging task. Computing the best ODT for a specific dataset has been proven an NP-complete problem [14], and algorithms based on exhaustive search exhibit a prohibitive computational complexity. As an example, CART-ELC [15] has a computational complexity equal to $O\left(\binom{|S|}{F} \cdot F^2(F+|S|)\right)$, where F is the number of features and $|S|$ is the size of the training set. Several heuristic procedures have been designed to derive the linear combination coefficients w_i at each node [16]–[21], but despite the advantages in model expressiveness, the large size of the parameter space often lead the training algorithms to produce overfitted models.

III. Shakan Methodology

While traditional axis-aligned DTs are efficient to evaluate, they are limited in modeling complex decision boundaries. ORFs, which use splits based on linear combinations of features, offer a valid alternative by enabling more expressive and compact models. They can capture richer patterns with fewer trees and shallower depths, thus having the potential to deliver higher accuracy with lower computational cost, provided that their execution can be optimized effectively. In this context, we develop an architecture that executes ORFs nodes with minimal overhead compared to traditional DT nodes. To reach this goal, we adapt the training algorithm to produce ORFs designed for efficient execution on the target architecture. This allows us to leverage the expressive power of ORFs to reach higher accuracy while inferring a more

compact model achieving lower latency. In this chapter, we detail Shakan training algorithm and hardware architecture.

A. Shakan Training Algorithm

RFs are trained by building multiple DTs, each using a different random subset of the training data. This is done through bootstrap sampling, where data points are randomly selected with replacement to create a unique training set for each DT. At the same time, random feature selection is applied at each node, meaning that only a small, random subset of the input features (usually $\sqrt{F}$ features, where F is the total number of features) is considered when deciding how to split the data. This technique, known as bagging, introduces variation across the DTs, helping reduce overfitting and increasing the stability of the model.

Shakan uses a different approach to reduce overfitting, regularizing split selection rather than performing bootstrap sampling. This technique, employed in other ML methods such as Extra-Trees [22], is particularly suited for ODTs, which make splits based on linear combinations of multiple features. Without regularization, such trees can create overly complex decision boundaries that do not generalize well to new data. Regularization is imposed by constraining oblique splits to at most three non-zero coefficients, meaning that each node only considers the value of three features. Moreover, we fix the first non-zero coefficient to one (without loss of generality), and we restrict the others to a limited set of values, composed by (signed) powers of 2:

$$\Gamma = \left\{0, -2^{-1}, -2^{0}, -2^{1}, 2^{-2}, 2^{-1}, 2^{0}, 2^{1}\right\}$$

This formulation not only mitigates overfitting, preventing excessively complex decision splits, but also significantly accelerates training while enabling efficient hardware inference. Indeed, multiplication by elements of Γ can be implemented in hardware by means of bitwise shifts and a sign flip, except in the trivial case of zero.

At each node, the training procedure draws $\sqrt{F}$ subsets of features of size 3 (out of $\binom{F}{3}$ possible subsets). For each subset, several linear combinations of the three features are tested, fixing the first coefficient to 1 and choosing the other two coefficients from Γ. Including 0 in this set allows the model to ignore a feature entirely, making it possible for ODTs to also represent standard axis-aligned splits. In this way, Shakan constitutes a *superset* of traditional DTs, combining their strengths with additional expressivity. Denoting as $\gamma_2, \gamma_3 \in \Gamma$ the two coefficients related to the second and third selected features, and with $\mathbf{f}(i)$ the i-th selected feature ($i \in \{1, 2, 3\}$), the final decision rule compares the linear combination of the selected features to a learned threshold:

$$X_{\mathbf{f}(1)} + \gamma_2 \cdot X_{\mathbf{f}(2)} + \gamma_3 \cdot X_{\mathbf{f}(3)} < \mathtt{th} \qquad (2)$$

Algorithm 1 shows the selection procedure applied at each node during the recursive construction of one ODT. The main loop in Lines 4-14 iterates $\sqrt{F}$ times, selecting a subset of features at each iteration, while the loop in Lines 6-13 iterates over *all* possible values of γ_2 and γ_3. The

Algorithm 1: SELECTNODEFEATURES

Input: S: array of samples,
F: number of features
Output: $\mathbf{f}$: selected features,
γ_2, γ_3: selected coefficients,
$\mathtt{th}$: selected threshold

1 $\mathtt{trials} \leftarrow 0$
2 $\mathtt{best_score} \leftarrow \infty$
3 $\mathbf{f}, \gamma_2, \gamma_3, \mathtt{th} \leftarrow \bot, \bot, \bot, \bot$ `//null values`
4 **while** $\mathtt{trials} < \sqrt{F}$ **do**
5 $\quad \widetilde{\mathbf{f}} \leftarrow \textsc{Sample}(\{0, 1, \ldots, F-1\}, 3)$
6 $\quad$ **for** $\widetilde{\gamma_2}, \widetilde{\gamma_3} \in \Gamma \times \Gamma$ **do**
7 $\quad\quad S_{\mathtt{sort}} \leftarrow \textsc{SortBy}(S, \widetilde{\mathbf{f}}, \widetilde{\gamma_2}, \widetilde{\gamma_3})$
8 $\quad\quad$ **for** i **from** 1 **to** $|S| - 1$ **do**
9 $\quad\quad\quad S_{\mathtt{L}}, S_{\mathtt{R}}, \widetilde{\mathtt{th}} \leftarrow \textsc{SplitAt}(S_{\mathtt{sort}}, \widetilde{\mathbf{f}}, \widetilde{\gamma_2}, \widetilde{\gamma_3}, i)$
10 $\quad\quad\quad \mathtt{score} \leftarrow \textsc{Score}(S_{\mathtt{L}}, S_{\mathtt{R}})$
11 $\quad\quad\quad$ **if** $\mathtt{score} < \mathtt{best_score}$ **then**
12 $\quad\quad\quad\quad \mathtt{best_score} \leftarrow \mathtt{score}$
13 $\quad\quad\quad\quad \mathbf{f}, \gamma_2, \gamma_3, \mathtt{th} \leftarrow \widetilde{\mathbf{f}}, \widetilde{\gamma_2}, \widetilde{\gamma_3}, \widetilde{\mathtt{th}}$
14 $\quad \mathtt{trials} \leftarrow \mathtt{trials} + 1$
15 **return** $\mathbf{f}, \gamma_2, \gamma_3, \mathtt{th}$

function $\textsc{Sample}(\mathtt{set}, \mathtt{n})$ at Line 5 samples $\mathtt{n}$ elements from $\mathtt{set}$ without repetition. The function $\textsc{SortBy}(S, \widetilde{\mathbf{f}}, \widetilde{\gamma_2}, \widetilde{\gamma_3})$ at Line 7 returns an array of samples sorted by the quantity on the left in Equation (2), and requires $\mathcal{O}(|S| \log |S|)$ steps. The loop in Lines 8-13 iterates over all the samples in the ordered dataset, performing a split each position and evaluating the quality of the resulting split. The function $\textsc{SplitAt}(S_{\mathtt{sort}}, \widetilde{\mathbf{f}}, \widetilde{\gamma_2}, \widetilde{\gamma_3}, i)$ at Line 9 assigns the first i samples to $S_{\mathtt{L}}$, and the other samples to $S_{\mathtt{R}}$, also returning the threshold $\widetilde{\mathtt{th}}$ that splits the dataset between the $(i-1)$-th sample and the i-th one. The $\textsc{Score}(S_{\mathtt{L}}, S_{\mathtt{R}})$ procedure at Line 10 returns an impurity score (e.g., Gini impurity) computed over the split dataset. Finally, the function SELECTNODEFEATURES returns the selected features $\mathbf{f}$, the two coefficients γ_2, γ_3, and the splitting threshold $\mathtt{th}$, that minimize the impurity of the split dataset. The computational complexity of Algorithm 1 is $\mathcal{O}\left(\sqrt{F} \cdot |\Gamma|^2 \cdot (|S| \log |S| + |S|)\right) = \mathcal{O}\left(\sqrt{F} |S| \log |S|\right)$ (since Γ is fixed), which is the same complexity of a split selection procedure in traditional DTs [23], and provides an exponential speedup over more complex ODT training procedures.

B. Shakan Hardware Architecture

Inspired by the architectural layout proposed by Moyogi [10], composed by multiple parallel pipelines of PEs, Shakan proposes a novel architecture specifically designed for the inference of ORFs.

In Shakan, each PE processes a sample in three pipelined clock cycles. In the first cycle, the PE fetches the node instruction from the ME. The second cycle is fully dedicated to the computation of the decision condition that is particularly more complex with respect to a standard DT, due to the

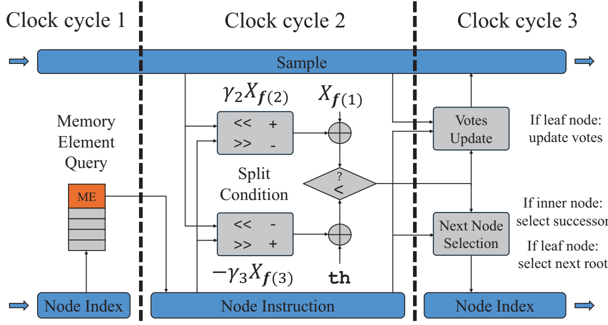

Fig. 2. PE structure in Shakan. Dashed lines separate operations executed in different clock cycles. Operands $<<$ and $>>$ denote left and right bitwise shift, while $<$ denotes numerical comparison.

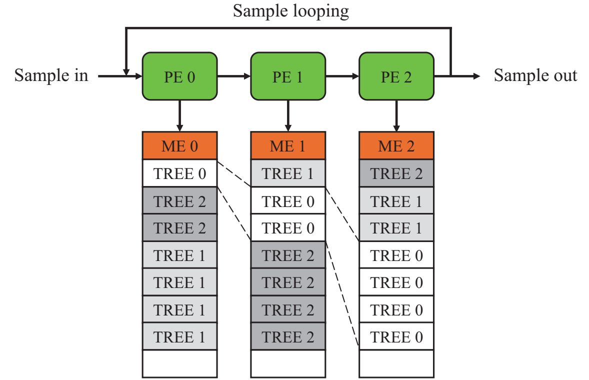

Fig. 3. Shakan basic-block architecture, containing three ODTs at depth 3.

usage of a linear combination of multiple features. These extra calculations can increase the circuit delay, potentially causing timing closure issues. From an operational standpoint, Equation (2) involves two bitwise shifts to compute $\gamma_2 \cdot X_{\mathbf{f}(2)}$ and $\gamma_3 \cdot X_{\mathbf{f}(3)}$, two sums and a comparison. To accelerate the computation in Equation (2), the two summations are performed in parallel by rewriting the equation as follows:

$$X_{\mathbf{f}(1)} + \gamma_2 \cdot X_{\mathbf{f}(2)} < \mathtt{th} - \gamma_3 \cdot X_{\mathbf{f}(3)} \tag{3}$$

Lastly, in the third clock cycle, based on the result of the splitting criterion, the PE determines which child node to visit next, and updates the class votes if the node is a leaf.

The resulting structure of each PE is shown in Figure 2. The PE receives as input one sample and the address of the current node instruction. During the first clock cycle, the PE queries the corresponding ME to read the node instruction. During the second clock cycle, the splitting decision is computed using Equation (3). During the third clock cycle, the PE behaves differently depending on the outcome of the splitting decision. If the successive node is an inner node, then its address is passed to the next PE in the pipeline. If the successive node is a leaf node, then the class vote is added to the sample, and the address passed to the next PE corresponds to the root node of the next tree to be executed. The PE shown in Figure 2 is fully pipelined, meaning that all the PEs handle three samples at each clock cycle, without impacting the overall throughput.

Another important architectural novelty involves the way node instructions are encoded and organized in the MEs. Shakan encodes an instruction in 63 bits, comprising:

- 16 bits for the (fixed point) threshold th;
- 1 validity bit;
- 1 bit for each child node to indicate whether it is a leaf node or an internal node;
- 6 bits to identify each of the three employed features ($\mathbf{f}(1)$, $\mathbf{f}(2)$, and $\mathbf{f}(3)$);
- 3 bits for each of the two coefficients (γ_2 and γ_3);
- 10 bits for each child node to indicate either the voted class or the address of the subsequent node instruction in the successive ME.

Comparing such instruction format to the one of Moyogi [10] for traditional DTs, only 15 additional bits are needed, 18 more to encode $\mathbf{f}(2)$, $\mathbf{f}(3)$, γ_2, and γ_3 but 3 less because depth information is omitted, since it is only required when MD-RF models are employed. However, due to the alignment with the 32-bit word size of each ME, both the instructions format are padded to 64 bits. In this way, the additional complexity needed to encode ODTs is completely hidden and no additional overhead is introduced in the instruction format.

As concerns the way instruction are stored in memory, in the architecture of Moyogi, each DT is mapped with the root in the first layer, forcing several DTs to be reduced in depth due to the limited size of the MEs, as shown in Figure 1. Moreover, several memory cells are not used, since adding too shallow DTs would negatively impact the accuracy. To address this, Shakan adopts a circulant memory mapping scheme, as illustrated in Figure 3. Each DT is stored across consecutive MEs, but if a root node starts near the end of the memory space, the successive nodes wrap around and continue from the first ME, allowing the DT to reach full depth.

This strategy brings two main advantages. First, it allows the architecture to fit full-depth ODTs, exploiting memory layout to avoid the need for a MD-RF. As a consequence, unlike the Moyogi architecture, Shakan does not employ a Neural Network (NN) for vote weighting. Removing the NN allows the training process to be fully parallelized, leading to a significant speedup during tasks such as hyperparameter tuning, model selection and feature selection. Furthermore, by avoiding the need for a separate validation set, Shakan can use the full dataset for building the trees, which can improve model accuracy, especially in settings where training data is limited. Second, the Shakan memory layout allows for a reduction in the number of unused memory cells. Indeed, in Moyogi there is a high memory utilization in the last MEs of each basic block, while the first layers are not fully utilized (Figure 1 being an example). In our architecture, memory utilization is spread across all the MEs, fully exploiting the available memory by reducing the amount of unused cells.

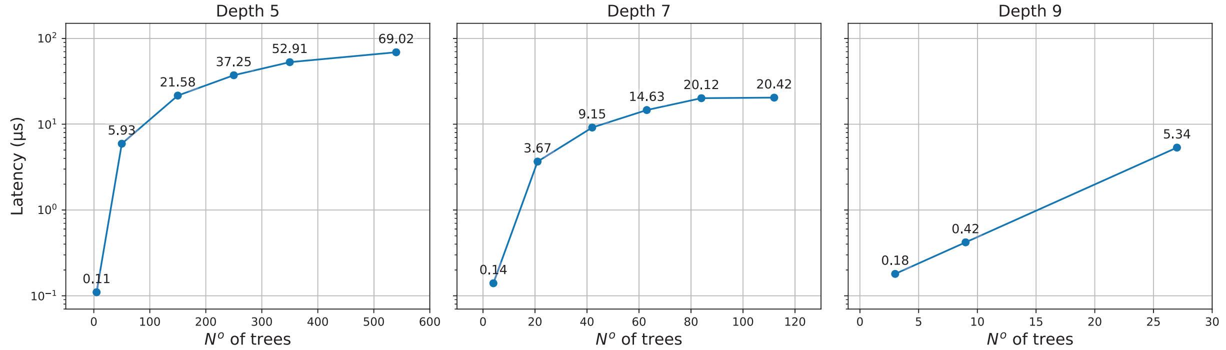

Fig. 4. Average latency of several configurations of Shakan at depths equal to 5, 7, and 9.

IV. Experimental Results

This Section details the experimental results of Shakan architecture, in terms of inference latency and classification accuracy on different datasets. We compare the achieved performance of our architecture to the performance of Moyogi, the current state-of-the-art solution for low-latency DT ensembles on embedded devices.

A. Experimental Setup

The logical core of the architecture has been developed in Chisel [24], using the Spatial Template Architecture Tool (SATL) [9]. The experiments have been executed on a Zynq UltraScale+ MPSoC mounted on a ULTRA96-V2-G embedded board, at a frequency of 166MHz. In Shakan, each sample contains five 32-bit fixed point features, 7 16-bit integer scores to accumulate the results, and other useful attributes to perform the computation, for an overall (padded) width of 320 bits. Moyogi requires 5 additional 16-bit fixed point weights to weigh the votes of the DTs, increasing the (padded) size of the sample to 384 bits. In both architectures, each ME is represented by a 36Kb BRAM, and each node instruction is 64 bits long. The architecture is controlled by a software host based on the PYNQ framework [25]. To assess and compare the accuracy of Moyogi and Shakan, we utilize five datasets relevant to classification tasks in real-world embedded applications: two datasets from [26] (Accelerometer, Person Activity) and three from [27] (Satellite, Vehicle, Vowel).

B. Latency and Accuracy Evaluation

Figure 4 shows the latency of Shakan, considering a maximal tree depth equal to 5, 7 ad 9, and different number of ODTs in the ensemble. For each depth, the largest number of ODTs corresponds to the largest configuration that can be implemented in the device without exceeding hardware resources, or introducing timing closure issues.

Figure 5 presents a detailed accuracy-latency comparison between Moyogi and Shakan on the five evaluation datasets, tackling different classification tasks. Each point in the figures represents a distinct ensemble configuration, varying in the number of DTs. Shakan **consistently achieves better accuracy** across different ensemble sizes, while maintaining **competitive latency**, in turn providing a reliable yet efficient tool for classification tasks on embedded devices.

The relative importance of latency and accuracy strongly depends on the deployment context. In the case of latency-critical applications, Shakan provides low-latency configurations in the order of 100-200ns. When comparing the smallest configurations of Shakan and Moyogi, our approach exhibits a comparable latency, while providing an average *improvement* in accuracy (defined as the absolute difference between accuracy values) equal to 11.85% for depth 5, 9.55% for depth 7 and 7.34% for depth 9.

In the case of accuracy-oriented applications, Shakan performs better than Moyogi across all configurations. When comparing the largest possible number of DTs in both architectures, Shakan achieves an average accuracy improvement equal to 5.41% for depth 5, 5.08% for depth 7, and 5.68% for depth 9, with lower or equal latencies.

In applications where high accuracy with low latency is desired, Shakan provides desirable classification accuracy even with the simplest configurations. Indeed, when comparing the smallest number of DTs in Shakan with the most accurate configurations in Moyogi, Shakan still achieves an accuracy similar to the one of Moyogi (with a small average improvement equal to 2.06% for depth 5, 0.23% for depth 7, and 0.014% for depth 9), while providing a notable latency speedup equal to $1140.45\times$ for depth 5, $213.69\times$ for depth 7, and $29.33\times$ for depth 9.

V. Conclusions

In this work, we present Shakan, an optimized approach to ORFs inference on embedded systems. We introduce a **novel training algorithm** for ORFs that maximizes the usage of training data, allows for parallelized training, enables **efficient hardware inference**, and **prevents overfitting** in cases of data scarcity. We propose a new hardware architecture that accommodates **large ensembles** with full-depth ODTs while **optimizing memory usage**. Experimental results show that Shakan outperforms Moyogi in terms of classification accuracy across different ensemble sizes, and provides an important speedups in terms of inference latency.

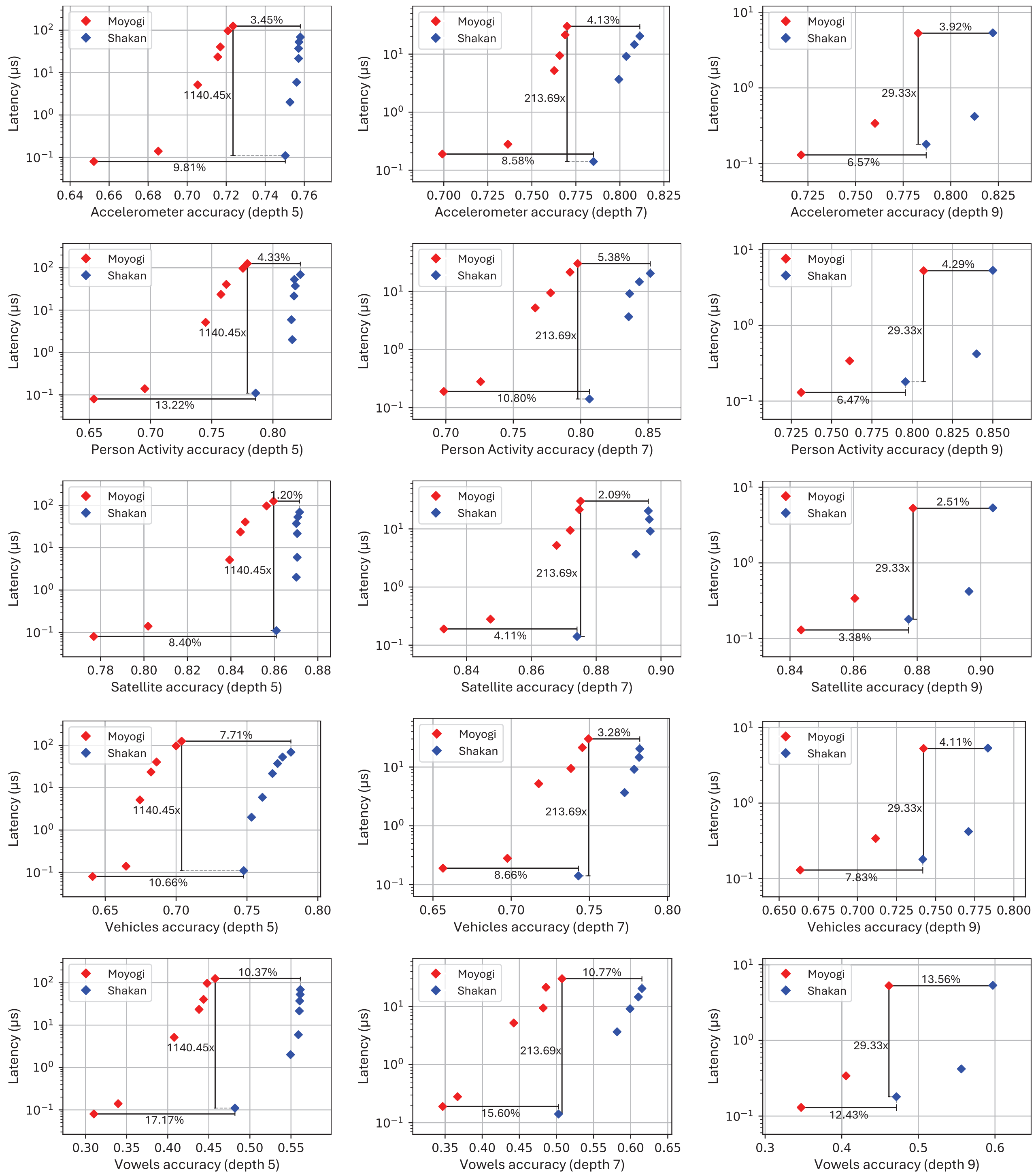

Fig. 5. Latency and accuracy comparison between Moyogi and Shakan on the five datasets. Each data point corresponds to a DTs configuration. The lowest horizontal bar shows the difference in accuracy between the smallest configurations in Shakan and Moyogi. The vertical bar shows the latency speedup between the largest configuration in Moyogi and the smallest configuration in Shakan. The highest horizontal bar shows the difference in accuracy between the largest configurations in Shakan and Moyogi.

REFERENCES

[1] H. Hua, Y. Li, T. Wang, N. Dong, W. Li, and J. Cao, "Edge computing with artificial intelligence: A machine learning perspective," *ACM Comput. Surv.*, vol. 55, no. 9, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3555802

[2] X. Lin, R. S. Blanton, and D. E. Thomas, "Random forest architectures on fpga for multiple applications," in *Proceedings of the Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 415–418. [Online]. Available: https://doi.org/10.1145/3060403.3060416

[3] B. H. Menze, B. M. Kelm, D. N. Splitthoff, U. Koethe, and F. A. Hamprecht, "On oblique random forests," in *Machine Learning and Knowledge Discovery in Databases*, D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 453–469.

[4] S. Summers, G. D. Guglielmo, J. Duarte, P. Harris, D. Hoang, S. Jindariani, E. Kreinar, V. Loncar, J. Ngadiuba, M. Pierini, D. Rankin, N. Tran, and Z. Wu, "Fast inference of boosted decision trees in fpgas for particle physics," *Journal of Instrumentation*, vol. 15, no. 05, p. P05026–P05026, May 2020. [Online]. Available: http://dx.doi.org/10.1088/1748-0221/15/05/p05026

[5] A. Damiani, E. D. Sozzo, and M. D. Santambrogio, "Large forests and where to "partially" fit them," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 550–555.

[6] T. D. Pham, C. Pham-Quoc, T. N. Thinh, B. K. do Nguyen, and C.-K. Pham, "A flexible and efficient fpga-based random forest architecture for iot applications," *Internet Things*, vol. 22, p. 100813, 2023.

[7] C. Pham-Quoc, "Scalable and efficient architecture for random forest on fpga-based edge computing." Berlin, Heidelberg: Springer-Verlag, 2023, p. 42–54.

[8] A. Verosimile, A. Tierno, A. Damiani, and M. D. Santambrogio, "Yoseue: "trimming" random forest's training towards resource-constrained inference," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 32–37.

[9] F. Peverelli, A. Verosimile, D. Conficconi, A. Damiani, and M. D. Santambrogio, "Satl: A spatial architecture rapid prototyping framework for irregular applications acceleration," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 442–445.

[10] A. Verosimile, F. Peverelli, and M. D. Santambrogio, " Moyogi: A Memory-Centric Accelerator for Low-Latency Random Forest Inference on Embedded Devices ," in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 189–197. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/FCCM62733.2025.00065

[11] J. Struharik, "Implementing decision trees in hardware," in *2011 IEEE 9th International Symposium on Intelligent Systems and Informatics*, 2011, pp. 41–46.

[12] P. Teodorovic and R. Struharik, "Hardware acceleration of sparse oblique decision trees for edge computing," *Elektronika ir Elektrotechnika*, vol. 25, no. 5, pp. 18–24, 2019.

[13] R. Choudhury, S. Rafi Ahamed, and P. Guha, "Simplified oblique decision tree accelerator," vol. 17, no. 2, p. 79–82, Oct. 2024. [Online]. Available: https://doi.org/10.1109/LES.2024.3475397

[14] D. Heath, S. Kasif, and S. Salzberg, "Induction of oblique decision trees," in *IJCAI*, vol. 1993. Citeseer, 1993, pp. 1002–1007.

[15] A. D. Laack, "Cart-elc: Oblique decision tree induction via exhaustive search," 2025. [Online]. Available: https://arxiv.org/abs/2505.05402

[16] E. Cantu-Paz and C. Kamath, "Inducing oblique decision trees with evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 1, pp. 54–68, 2003.

[17] R. C. Barros, M. P. Basgalupp, A. C. P. L. F. de Carvalho, and A. A. Freitas, "A survey of evolutionary algorithms for decision-tree induction," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 3, pp. 291–312, 2012.

[18] F. E. Otero, A. A. Freitas, and C. G. Johnson, "Inducing decision trees with an ant colony optimization algorithm," *Applied Soft Computing*, vol. 12, no. 11, pp. 3615–3626, 2012. [Online]. Available: https://doi.org/10.1016/j.asoc.2012.05.028

[19] R. Struharik, V. Vranjković, S. Dautović, and L. Novak, "Inducing oblique decision trees," in *2014 IEEE 12th International Symposium on Intelligent Systems and Informatics (SISY)*, 2014, pp. 257–262.

[20] F. Bollwein, "A pivot-based simulated annealing algorithm to determine oblique splits for decision tree induction," *Comput. Stat.*, vol. 39, no. 2, pp. 803–834, 2024. [Online]. Available: https://doi.org/10.1007/s00180-022-01317-1

[21] D. C. Wickramarachchi, B. L. Robertson, M. Reale, C. J. Price, and J. Brown, "HHCART: an oblique decision tree," *CoRR*, vol. abs/1504.03415, 2015. [Online]. Available: http://arxiv.org/abs/1504.03415

[22] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006.

[23] H. M. Sani, C. Lei, and D. Neagu, "Computational complexity analysis of decision tree algorithms," in *Artificial Intelligence XXXV*, M. Bramer and M. Petridis, Eds. Cham: Springer International Publishing, 2018, pp. 191–197.

[24] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.

[25] PYNQ, "Pynq framework documentation." [Online]. Available: https://www.pynq.io/

[26] C. L. Blake and C. J. Merz, *UCI Repository of Machine Learning Databases*, University of California, Irvine, Department of Information and Computer Sciences, Irvine, CA, 1998.

[27] F. Leisch and E. Dimitriadou, *mlbench: Machine Learning Benchmark Problems*, 2024. [Online]. Available: https://CRAN.R-project.org/package=mlbench