

Towards Efficient and Scalable Acceleration of Online Decision Tree Learning on FPGA

Zhe Lin*, Sharad Sinha[†] and Wei Zhang*

*Hong Kong University of Science and Technology, Hong Kong

[†]Indian Institute of Technology (IIT) Goa, India

zlinaf@ust.hk, sharad_sinha@ieee.org, wei.zhang@ust.hk

Abstract—Decision trees are machine learning models commonly used in various application scenarios. In the era of big data, traditional decision tree induction algorithms are not suitable for learning large-scale datasets due to their stringent data storage requirement. Online decision tree learning algorithms have been devised to tackle this problem by concurrently training with incoming samples and providing inference results. However, even the most up-to-date online tree learning algorithms still suffer from either high memory usage or high computational intensity with dependency and long latency, making them challenging to implement in hardware. To overcome these difficulties, we introduce a new quantile-based algorithm to improve the induction of the Hoeffding tree, one of the state-of-the-art online learning models. The proposed algorithm is light-weight in terms of both memory and computational demand, while still maintaining high generalization ability. A series of optimization techniques dedicated to the proposed algorithm have been investigated from the hardware perspective, including coarse-grained and fine-grained parallelism, dynamic and memory-based resource sharing, pipelining with data forwarding. We further present a high-performance, hardware-efficient and scalable online decision tree learning system on a field-programmable gate array (FPGA) with system-level optimization techniques. Experimental results show that our proposed algorithm outperforms the state-of-the-art Hoeffding tree learning method, leading to 0.05% to 12.3% improvement in inference accuracy. Real implementation of the complete learning system on the FPGA demonstrates a 384× to 1581× speedup in execution time over the state-of-the-art design.

I. INTRODUCTION

Decision tree algorithms are a popular class of machine learning algorithm and have been deployed in many real scenarios [1]–[3], especially when multiple decision trees are combined into powerful ensemble models, such as XG-Boost [4] and random forests [5]. Recently, the ensemble of decision trees as deep forests [6] has been reported to produce comparable performance compared to deep neural networks. However, there are several drawbacks that limit the full exploitation of the traditional decision trees (e.g., IDT3 [7], CART [8] and C4.5 [9]). The first drawback is the extensive memory consumption during the training process, which is proportional to the size of datasets. Classic decision tree learners assume that the complete datasets can be preloaded before training starts. This reduces their capability to train with large-scale datasets, especially when, nowadays, large amount of data is being generated daily. The second disadvantage comes with the learners’ inability to adapt themselves to new data once the training process is terminated. In the era of

big data, the size of datasets is no longer the bottleneck of learning algorithms. Instead, the ability to effectively learn from massive data and rationally make use of incoming data becomes more fundamental and critical.

To broaden the applicability of decision tree algorithms, extensions from traditional tree algorithms to batch learning and online learning (or so-called incremental learning) have been studied, which aim at adapting the models to incoming data without losing previously learned knowledge. One of the state-of-the-art online learning methods for streaming data is the *Hoeffding tree* [10] algorithm and its variants [11]–[18]. The Hoeffding tree presents an enhancement of the decision tree induction algorithm which leverages the accumulated samples to estimate complete datasets statistically. It is capable of performing training and inference concurrently. The Hoeffding tree is widely used in various application scenarios [19]–[22].

While efficient software implementation has been investigated for processors to accelerate the Hoeffding tree [12], [13], there are still many hindrances to the compact implementation and optimization of the Hoeffding tree design from the hardware perspective. We identify two principal challenges limiting Hoeffding tree implementation in hardware: 1) the high cost of memory usage to store the required subset of samples as well as characteristics in each leaf node; and 2) the high computational demand with dependency and long latency between iterations in the learning process, which can hamper efficient data processing with optimization schemes such as parallelism and pipelining. Furthermore, we observe a trade-off between the above two factors in the state-of-the-art designs: the methods in [13] and [14], attempting to reduce the memory usage, tend to extensively increase the computational intensity and latency, and vice versa, as in the proposed methods of [11] and [12]. The high and unbalanced need of memory and computation makes the existing approaches difficult to efficiently implement in hardware, especially on FPGAs where memory and digital signal processing (DSP) resources are both limited. Motivated by the above challenges and observations, we seek opportunities to implement and optimize the Hoeffding tree in a hardware-friendly and scalable way, and also strive to make use of resources in a more balanced manner. In this paper, we propose the first and complete implementation of the Hoeffding tree learning system on FPGA, with the following contributions:

- We first introduce a quantile-based algorithm for Hoeffding

ing tree induction, which uses light-weight computation and constant memory, while preserving high accuracy.

- We present hardware optimization techniques dedicated to the proposed algorithm, in order to achieve high hardware efficiency and scalability. These includes different levels of parallelism, dynamic and memory-based resource sharing, and pipelining with data forwarding.
- We investigate optimization techniques for tree growing, categorical attribute learning and split judgment to establish the complete online decision tree system on FPGA.

II. ALGORITHM AND CHALLENGES

A. Hoeffding Tree Induction Algorithm

The basic induction flow of the Hoeffding tree [10] is the same as the conventional decision trees [23], except that Hoeffding tree exploits the potential for the currently seen sample set to represent an infinite sample set. The Hoeffding tree algorithm is described in Algorithm 1. *Hoeffding bound* (additive Chernoff bound) [24] tells how close the current best split approaches the optimal split given an infinite sample set. Suppose we make n independent observations of a random variable r within range R . The Hoeffding bound guarantees that the true mean \bar{r} of r will be at least $E[r] - \epsilon$, with

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}. \quad (1)$$

Let $G(a_i)$ be the best measurement (e.g., gini impurity reduction) of a chosen split attribute a_i . The Hoeffding tree searches for the best and second-best $G(\cdot)$ values amongst all attributes. Given the sample set of size n for a specific node and a desired δ , the Hoeffding bound justifies that the current best attribute is the exact best attribute from an infinite dataset with probability $1 - \delta$, if the following equation is satisfied:

$$G(\text{Best attr.}) - G(2^{nd} \text{ Best attr.}) > \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}. \quad (2)$$

An additional tie condition is applied: when the two best attributes have close $G(\cdot)$, a split is taken if the Hoeffding bound is lower than a certain threshold τ . That is,

$$G(\text{Best attr.}) - G(2^{nd} \text{ Best attr.}) < \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} < \tau. \quad (3)$$

B. Challenges

Studies [11]–[14] have introduced several methods to improve the basic Hoeffding tree algorithm. These methods, however, reveal two main challenges for hardware implementation.

1. High Cost of Memory Utilization. In VFML [12], both numeric and categorical attribute values are preserved in a fixed number of bins (denoted as n_{ijk}) in a first-come-first-served manner. If all the bins are occupied, the newly coming attribute values unseen in all the bins are simply discarded during runtime. Although this method works well with categorical attributes of which values are discrete and the total number can be determined in the compile time, it requires a bin of large size to fit each numeric attribute per class per node to achieve a wide value coverage. Hence, the memory

Algorithm 1: Traditional Hoeffding tree algorithm

```

input : samples denoted as  $(x, y)$ 
output: Hoeffding tree denoted as  $HT$ 
1 for each  $(x_t, y_t)$  coming at time  $t$  do
2   filter  $(x_t, y_t)$  to leaf  $l$  of  $HT$ 
3   sample number in leaf  $l$ :  $n_l \leftarrow n_l + 1$ 
4   update bin count  $(attr_i, val_j, class_k)$   $n_{ijk}$  in leaf  $l$ 
5   if split trial is activated then
6     compute left/right partitions according to  $n_{ijk}$ 
7     compute  $G(\cdot)$  for each attribute
8     if  $G(\text{best}) - G(2^{nd} \text{ best}) > \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$  or
9        $\sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}} < \tau$  then
10      Split leaf  $l$  on the best attr.
        Initialize count  $n_{ijk}$  for each leaf

```

Algorithm 2: Incremental Gaussian approximation

```

input : samples denoted as  $(attr_{val}, weight)$ 
output: mean of Gaussian approximation denoted as  $M$ 
output: variance of Gaussian approximation denoted as  $V$ 
1 weight sum:  $w\_sum \leftarrow \text{first weight}$ 
2 variance sum:  $v\_sum \leftarrow 0$ 
3  $M \leftarrow \text{first } attr_{val}$ 
4 for each sample  $(attr_{val}, weight)$  in sample set do
5    $w\_sum \leftarrow w\_sum + weight$ 
6    $M_{prior} \leftarrow M$ 
7    $M \leftarrow M + \frac{attr_{val} - M_{prior}}{w\_sum}$ 
8    $v\_sum \leftarrow v\_sum + (attr_{val} - M_{prior}) \times (attr_{val} - M)$ 
9    $V \leftarrow \frac{v\_sum}{w\_sum - 1}$ 

```

requirement grows significantly with the number of attributes. This similarly exists in the method [14] using Greenwald and Khanna summaries [25], which requires to construct sample distribution from up to thousands of tuples per attribute-class combination per node. The exhaustive binary tree method [11] also suffers from injudicious use of memory because it needs to dynamically allocate memory for sample storage.

2. High Computational Intensity with Dependency and Long Latency. To reduce memory utilization, Gaussian-based methods [13], [14] are applied to trade much higher computational intensity for memory efficiency. For each numeric attribute per class, the sample distribution is estimated in a form of Gaussian distribution. As the Gaussian function is determined by only two values, namely, mean and variance, the memory usage can be significantly compressed to $\#attribute \times \#class \times 2$ per node. However, the incremental update process of the mean and variance leads to high computational demand, as shown in Algorithm 2. The requirement of computation resources is proportional to both the number of attributes and classes. Besides this, the split judgment stage also requires computing the cumulative density functions (CDFs) at each split point, which entails even higher computational power. Moreover, the update process incurs long latency and should be in order of time if the two successive iterations work on the same label. In addition to the high computational intensity, the long latency and data dependency further hinder this method from being effectively optimized in hardware.

III. METHODOLOGY

As BRAM and DSP are limited resources for FPGAs, the excessive use of either on-chip memory or computation units

Algorithm 3: Hoeffding tree induction with quantiles

input : streaming samples denoted as (x, y)
output: Hoeffding tree structure denoted as HT
 Let a_i ($1 \leq i \leq |A|$) denote the attribute in set A
 Let c_j ($1 \leq j \leq |C|$) denote the class in set C
 Let α_k ($1 \leq k \leq |Q|$) denote the quantile index

```

1 for each  $(x_t, y_t) \in \text{sample set}$  do
2   filter  $(x_t, y_t)$  to leaf  $f$  of  $HT$ 
3   sample num. at  $f$ :  $n_f \leftarrow n_f + 1$ 
4   for  $j$  from 1 to  $|C|$  do
5     sample num. in class  $j$ :  $n_{fj} \leftarrow (y_t == j) ? n_{fj} + 1 : n_{fj}$ 
6     for  $i$  from 1 to  $|A|$  do
7       max. attr. value:  $max_{a_i} \leftarrow (a_i > max_{a_i}) ? a_i : max_{a_i}$ 
8       min. attr. value:  $min_{a_i} \leftarrow (a_i < min_{a_i}) ? a_i : min_{a_i}$ 
9       for  $j$  from 1 to  $|C|$  do
10        if  $y_t == j$  then
11          for  $k$  from 1 to  $|Q|$  do
12             $Q_{ijt}(\alpha_k) \leftarrow$ 
13               $Q_{ijt-1}(\alpha_k) - \lambda sgn_\alpha(Q_{ijt-1}(\alpha_k) - a_i)$ 
14        if split trial is activated then
15          for  $i$  from 1 to  $|A|$  do
16            for  $p$  from 1 to  $|P|$  do
17               $pt \leftarrow \frac{max_{a_i} - min_{a_i}}{|P|+1} \times p + min_{a_i}$ 
18              for  $j$  from 1 to  $|C|$  do
19                left distribution  $L$ :  $dist_{Lij}(pt) \leftarrow 0$ 
20                for  $k$  from 1 to  $|Q|$  do
21                   $dist_{Lij}(pt) \leftarrow (pt >$ 
22                     $Q_{ijt}(\alpha_k)) ? dist_{Lij}(pt) + 1 :$ 
23                     $dist_{Lij}(pt)$ 
24                   $dist_{Lij}(pt) \leftarrow \frac{dist_{Lij}(pt)}{|P|} \times n_{fj}$ 
25                   $dist_{Rij}(pt) \leftarrow n_{fj} - dist_{Lij}(pt)$ 
26                compute  $G(a_i)$  for all  $pt$ 
27            if  $G(best) - G(2^{nd} \text{ best}) > \sqrt{\frac{R^2 \ln(1/\delta)}{2n_f}}$  or
28               $\sqrt{\frac{R^2 \ln(1/\delta)}{2n_f}} < \tau$  then
29              split  $l$  on the best attr & initialize new leaves

```

in the aforementioned methods [11]–[14] is neither efficient nor scalable while handling numeric attributes. The two design challenges described above and their interplay should be taken into consideration for joint optimization. To this end, we propose to introduce an up-to-date quantile algorithm in the induction of online decision trees.

A. Quantile Estimation Using Asymmetric Signum Functions

Quantiles [26] are cutting points dividing the range of a probability distribution into a certain number of intervals with equal probabilities. The quantile function $Q(\cdot)$ of a continuous variable is defined as the inverse of the CDF, $F(z) = Pr(x_t \leq z)$. Specifically, $Q(\cdot)$ can be written as

$$Q(\alpha) = F_X^{-1}(\alpha) = \inf\{x \in \text{supp}(F_X) : \alpha \leq F_X(x)\}. \quad (4)$$

The state-of-the-art quantile estimation using asymmetric signum functions is studied in [27] and [28]. The quantile approximation calibrates the quantiles in a sequential manner according to every incoming sample. The quantile calibration process from sample x_{t-1} to x_t can be described as

$$Q_t(\alpha) = Q_{t-1}(\alpha) - \lambda sgn_\alpha(Q_{t-1}(\alpha) - x_t), \quad (5)$$

where $sgn_\alpha(\cdot)$ is the asymmetric signum function defined by

$$sgn_\alpha(z) = \begin{cases} \alpha, & \text{if } z < 0 \\ 1 - \alpha, & \text{if } z \geq 0 \end{cases}. \quad (6)$$

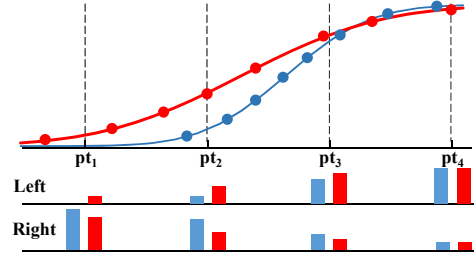


Fig. 1. Partition strategy in the proposed algorithm, illustrated with one attribute, two labels and eight quantiles.

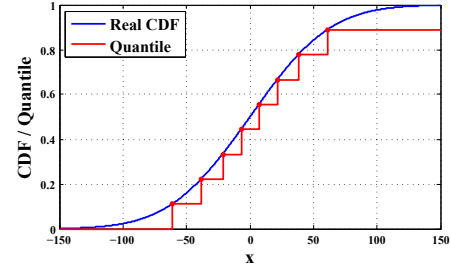


Fig. 2. Using eight quantiles to estimate the CDF of normal distribution with a round-down scheme.

B. Learning Numeric Attributes with Quantile Approximation

To handle numeric attributes, we develop a new algorithm in the Hoeffding tree induction process by applying the quantile estimation with asymmetric signum functions, which is described in Algorithm 3. The proposed algorithm encompasses two key features: 1) a separate set of quantiles is maintained per attribute per class (line 6 to 12); and 2) the strategy to get left/right partitions based on the attribute distributions (line 14 to 22) has been customized to support the quantile method. Note that the number of quantiles to use is determined by the characteristics of the datasets. This is studied in Section V-B.

A straightforward method [12] to deduce the partitions is to view each sample as a split point and compute distribution individually: for an attribute i and a specific sample's attribute as the split point pt_i , an arbitrary sample is sorted to the left partition if its attribute value $a_i \leq pt_i$, or otherwise, it is filtered to the right partition. In our algorithm, we learn the samples with quantiles and represent sample distribution in CDF: each quantile value $Q(\alpha_k)$ indicates that the percentage is α_k for the samples with the attribute values smaller than $Q(\alpha_k)$. In this way, sample storage is not required.

Fig. 1 illustrates how the overall partitioning strategy works. We generate a set of split points evenly distributed in the full range of attribute values. These split points are compared to the quantiles individually to find out the interval of two quantiles $[Q(\alpha_k), Q(\alpha_{k+1})]$ containing the split point. Afterwards, the sample number in each partition can be determined. The portion of samples with attribute values smaller than or equal to $Q(\alpha_k)$ goes to the left partition, whereas the others go to the right partition. By this method, the sample distribution in the left partition is rounded down to the nearest quantile, with an example shown in Fig. 2.

The proposed algorithm overcomes the trade-off between memory and computation, and presents a more rational and balanced solution compared with state-of-the-art meth-

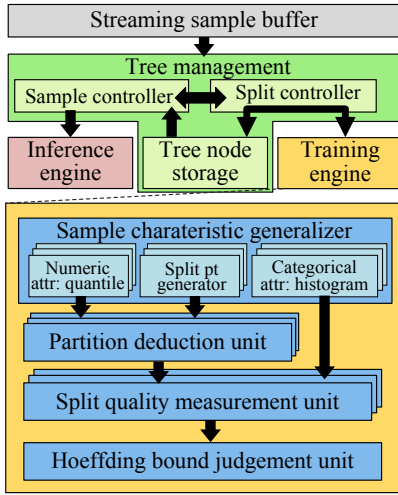


Fig. 3. System overview of the Hoeffding tree implementation with the proposed algorithm.

ods [11]–[14]. The advantages of this proposed method are three-fold. Firstly, the sample characteristics are fully generalized and encapsulated in a set of quantiles, dispensing with the need to store any samples in the training iterations. The memory requirement is reduced to $\#attribute \times \#label \times \#quantile$ per leaf node. This outperforms existing methods [11], [12] which require large attribute or sample storage. Secondly, the computation demand is notably reduced compared with the memory-efficient yet computation-intensive method, Gaussian method [13], [14]: only comparison and subtraction are involved in quantile approximation, whereas Gaussian approximation entails expensive computation as shown in Algorithm 2. The complexity of partition deduction is also effectively simplified with the proposed method. Thirdly, the problem of data dependency can be resolved with hardware optimization through deliberate parallelism and pipelining, as introduced in Section IV-C.

IV. ARCHITECTURE DESIGN

A. System Overview

The system overview of the Hoeffding tree implementation is depicted in Fig. 3. Starting from the sample buffer, the tree management engine first reads and decodes the sample information. At the same time, it fetches relevant tree nodes from the tree node storage and filters the samples to the leaf nodes in a pipelined way. Thereafter, both the inference engine and training engine start processing the samples.

In the learning process, samples are decomposed into separate attributes and the characteristics of each attribute are learned and stored independently. When a split trial is invoked at a leaf node, for each numeric attribute, a partition deduction unit uses the quantiles and split points to deduce left and right partitions. As for categorical attributes, the sample counts of all attribute-class combinations form a histogram, which is similar to the quantiles for numeric attributes.

The partition information of every attribute is then processed by a split quality measurement unit to compute the split gain for each split point. Then, the best and second-best split gains

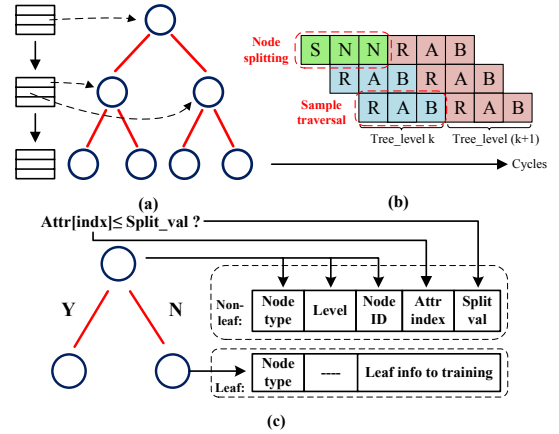


Fig. 4. (a) Decision tree architecture; (b) tree management pipeline structure; and (c) bit decomposition of tree node memory.

are identified, and the split decision is given by the Hoeffding bound judgment unit. If a split is taken, the split information is sent back to the split controller to update the tree structure.

B. Tree Management Units

The tree management units maintain two operations: 1) filtering samples to different leaf nodes, which requires tree traversing; and 2) splitting leaf nodes by overwriting the tree node memory after receiving split requests.

The tree traversing process for each sample starts from the root node down to a specific leaf node, thus involving several rounds of memory reading. Considering the case of streaming data input, the tree memory may receive multiple read requests from different samples concurrently. Multi-port memory can be used to support this feature. However, the required port number is linearly related to the tree depths. FPGA BRAMs naturally support up to two ports, and increasing the port size turns out to be an inefficient solution. We observe that the samples are processed at different tree levels sequentially and the samples from different time steps require memory reading from different tree levels. Hence, we separate the node storage according to tree levels, as depicted in Fig. 4 (a), and dual-port memory is enough to support both node splitting and tree traversing for streaming samples. The idea of using a separate memory structure has been adopted in DT-CAIF [29], whereas we develop a fine-grained pipeline structure for each tree level. All the tree levels together form a deep pipeline.

The fine-grained pipeline needs to support both tree traversing and node splitting. A three-stage pipeline is formed, as shown in Fig. 4 (b). The tree traversing routine consists of node reading (R), attribute selection (A) and branch decision (B) stages. As for node splitting, split information (mainly the split node level, node ID, split coefficient and attribute index) from the training engine is passed across different tree levels. When a leaf node is reached, the corresponding memory element is overwritten by the split information to replace the leaf node with an internal node. Moreover, two new leaf nodes are generated in the next level and the split pipeline also writes in the new leaf nodes the training elements they are associated to. This is related to the dynamic leaf node-element allocation

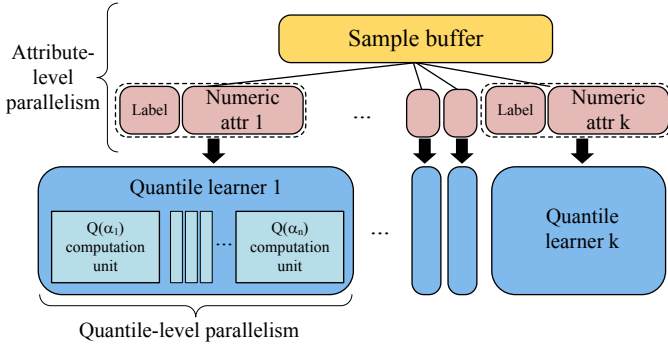


Fig. 5. Exploiting attribute-level and quantile-level parallelism.

scheme discussed in Section IV-C. All the operations relevant to the split are completed in the split (S) stage, after which two nop (N) states are followed. The bit information stored in the memory for branch decision is shown in Fig. 4 (c).

C. Learning Numeric Attributes

In our proposed Algorithm 3, recall that we maintain a set of quantile values per numeric attribute per class for a single leaf node. Optimization techniques are investigated for accelerating quantile learning from the hardware perspective, which can be summarized as: 1) attribute-level (coarse-grained) and quantile-level (fine-grained) parallelism; 2) dynamic and memory-based resource sharing; and 3) pipelining with data forwarding for data dependency removal.

Attribute-level (Coarse-grained) and Quantile-level (Fine-grained) Parallelism. As shown in line 6 to line 12 of Algorithm 3, different attributes are independent and, within each attribute, the quantiles $Q(\cdot)$ per class are also independent of each other. This allows us to speed up the quantile computation process with both attribute-level and quantile-level parallelism, as shown in Fig. 5. Note that we do not take class-level parallelism even though it is possible. This is because each sample contains a unique class label but has multiple attributes. The learning process only needs to update the set of quantiles matching the sample label. Based on this fact, parallelizing at class level does not offer any benefit. Instead, we seek opportunities for class-level optimization through resource sharing and pipelining.

Dynamic and Memory-based Resource Sharing. For each leaf node, it is required to maintain a number of quantiles per attribute per class. If hardware copies are simply replicated for each leaf node, both the memory and arithmetic resource utilization becomes too expensive for hardware to implement. In light of this problem, we develop a dynamic leaf node-element allocation scheme as the tree grows dynamically and a memory-based resource sharing mechanism for quantile update routine.

To differentiate between a leaf node of the tree and the physical resource allocated for a leaf node in the training process, we call the former a *leaf node*, while we denote the latter as an *element*. A leaf node is only temporarily being a leaf node, and it may be split as samples assemble. Therefore, it is not necessary to statically allocate physical resources

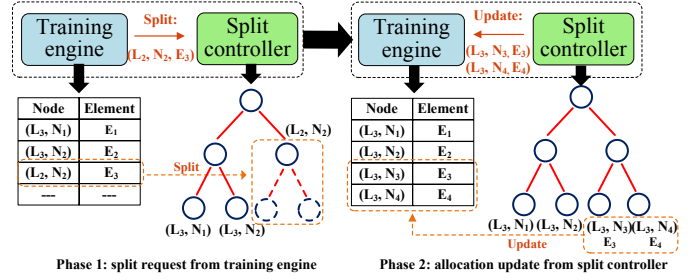


Fig. 6. Dynamic leaf node-element allocation scheme.

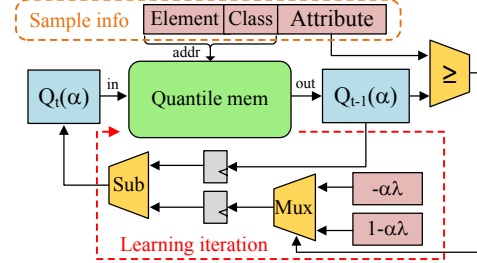


Fig. 7. A single quantile computation unit with the memory-based resource sharing scheme.

to each leaf node. We devise a *dynamic leaf node-element allocation scheme*, as shown in Fig. 6. The training engine maintains a node-element table to keep track of the leaf node-element pairs. During the split process, the split controller generates new leaf node-element pairs and sends them back to the training engine. The training engine then updates the leaf node-element relationship in the table. In this way, the leaf node-element allocation change dynamically and resource reuse in hardware is facilitated.

A *memory-based resource sharing scheme* is designed to collaboratively work with the dynamic leaf node-element allocation scheme for further resource sharing. This scheme leverages two facts: 1) each sample is only sorted to one leaf node, so only one element will be activated for quantile update per sample; and 2) for each attribute, only the set of quantiles corresponding to the sample label will be activated per sample. Since the quantile learning process is the same for all classes and elements, except that the quantile values are different, we devise the following memory-based resource sharing scheme: for each attribute, all the classes of all elements share one set of quantile computation logics and all the corresponding quantile values are stored in one memory. When a sample is used for training, the set of quantiles corresponding to the specific element and class is fetched, and later, the updated values are stored back to the same memory location. Element and class values together form the memory addresses. Putting it all together, a single quantile computation unit with memory-based resource sharing is depicted in Fig. 7. To support this mechanism, each leaf node in the tree memory preserves a field denoted as *leaf information to training* shown in Fig. 4 (c). Provided a new split, the two new leaf nodes along with their assigned element IDs are sent from the training engine to the tree node memory for update. For each sample after tree traversing, the element ID associated with its reached leaf node and the raw sample data are sent to the training engine.

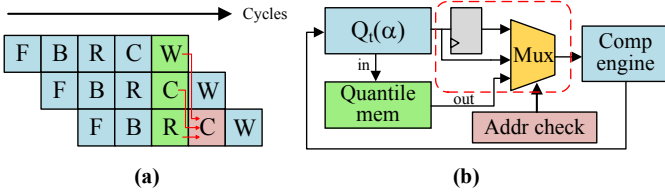


Fig. 8. (a) Pipelining stages with data forwarding; (b) hardware realization of data forwarding.

Pipelining with Data Forwarding. There exists data dependency for quantile computation: two successive samples sorted to the same leaf node should update the same element in a sequential way. For the method with Gaussian approximation, the long latency of the update process, as described in Algorithm 2, makes it difficult to overcome this dependency. For the proposed quantile computation architecture in Fig. 7, the computation is reduced to a comparison and a subtraction per quantile unit, which allows us to fully exploit the pipeline architecture with data forwarding to resolve data dependency.

We propose a 5-stage pipeline architecture for the quantile update routine, as shown in Fig. 8 (a). The first stage (F) fetches a sample from the sample buffer. The second stage (B) decides on the execution branch to take, including element initialization in the dynamic leaf node-element allocation scheme, quantile computation and quantile output for the split trial. In the next stage (R), the quantile unit selected by the element and class is read out. Afterwards, the quantile is updated in the computation stage (C) following Equation (5), and is written back to the same memory location in the writing stage (W).

In stage C, we address the data dependency problem by the adoption of a dedicated data forwarding method, as shown in Fig. 8 (b), which aims at providing the flexibility that, when the quantiles are updated while not yet written in the memory, they are directly passed to the quantile computation engine if the addresses between these two computation periods match. We keep track of the results and quantile memory addresses of the prior two computation periods, which are managed by stage C and stage W, respectively. Stage C has a higher forwarding priority over stage W when both memory addresses match the one currently processing, because stage C provides the most up-to-date results. This data forwarding allows us to bypass memory operations when dependency occurs and eventually leads to a throughput of one sample per cycle.

D. Learning Categorical Attributes

The process of learning categorical attributes is similar to learning numeric attributes. However, the value and size of each categorical attribute is determined by dataset characteristics, which can be known in design time. Therefore, counting the number of occurrence for each attribute-class combination gives a histogram of the distribution without any loss of information. In a split trial for categorical attributes, each attribute value is used as a split point individually: the samples with the attribute value equal to the split point is filtered to the left, or otherwise, it is sorted to the right.

The optimization methods, except the dynamic leaf node-element allocation scheme, can be migrated to categori-

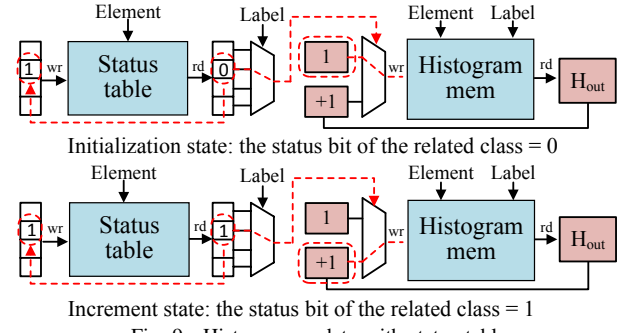


Fig. 9. Histogram update with status table.

cal attributes seamlessly. However, to support the dynamic leaf node-element allocation scheme, the histograms of all attribute-class combinations for an element need to be initialized simultaneously. This brings difficulties as we apply memory-based resource sharing in which the same dual-port histogram memory is shared amongst different class labels, and multiple write requests to the same memory is inefficient for FPGA design. To overcome this problem, we additionally implement a status table for histograms. Every memory unit in the status table represents an individual histogram, and each bit indicates the status of a column of this histogram. To initialize a histogram when a new leaf node-element pair is assigned, only the corresponding memory unit in the status table, instead of all units in the histogram, needs to be reset. The training routine first checks the status table for each incoming sample, and follows either of the two situations (i.e., initialization or increment) as depicted in Fig. 9. The relevant status bit is set to high when the first sample comes after initialization.

E. Simplification of Split Measurement with Hoeffding Bound

The study in [30] has shown that the choice of split measurement method does not exert a significant impact on the accuracy of decision tree induction. We adopt gini impurity [8] as it is commonly used and has low computational demand.

Gini impurity is a measure of the chance for an example to be incorrectly classified if it is randomly labeled according to the distribution of the labels. Let p_j be the probability of examples being labeled as class j ($j \in 1, 2, \dots, |C|$) in the dataset S . Gini impurity can be represented as

$$gini(S) = 1 - \sum_{j=1}^{|C|} p_j^2. \quad (7)$$

The split quality for a given partition is based on the reduction in gini impurity after a split is taken. If S is split into the left subset S_L and right subset S_R , the reduction in gini impurity can be described by

$$G = \Delta gini = gini(S) - \frac{|S_L|}{|S|} gini(S_L) - \frac{|S_R|}{|S|} gini(S_R). \quad (8)$$

We combine the split measurement with the Hoeffding bound judgment for joint optimization in hardware. Let $S_{L,j}$ and $S_{R,j}$ be the subset of S_L and S_R labeled in j , respectively. We reorganize the reduction in gini impurity G as follows:

$$G = \frac{1}{|S|} \left(\frac{1}{|S_L|} \sum_{j=1}^{|C|} |S_{L,j}|^2 + \frac{1}{|S_R|} \sum_{j=1}^{|C|} |S_{R,j}|^2 \right) + gini(S) - 1. \quad (9)$$

TABLE I
INFERENCE ACCURACY USING DIFFERENT NUMBERS OF QUANTILES.

Dataset	Gaussian method	Quantile method with different quantile size									
		2	4	8	16	24	32	64	128	256	512
Bank	89.10%	88.79%	89.05%	89.15%	89.30%	89.31%	89.32%	89.26%	88.52%	88.66%	88.59%
Telescope	76.16%	76.68%	74.61%	76.41%	76.12%	76.15%	76.64%	75.51%	75.75%	76.75%	71.32%
Electricity	76.26%	76.97%	77.26%	78.02%	76.31%	77.93%	77.53%	76.91%	76.75%	76.61%	74.15%
Covertypes	71.02%	72.46%	72.17%	72.72%	72.51%	72.50%	71.86%	73.43%	71.90%	70.94%	69.41%
Person	39.00%	45.90%	48.82%	51.38%	52.49%	53.37%	52.35%	52.40%	47.94%	47.44%	49.60%

Putting the gini impurity reduction and Hoeffding bound together, the calculation can be reorganized as

$$G_{B_1} - G_{B_2} = \frac{1}{|S|} \left[\underbrace{\left(\frac{1}{|S_{B_1,L}|} \sum_{j=1}^{|C|} |S_{B_1,L,j}|^2 + \frac{1}{|S_{B_1,R}|} \sum_{j=1}^{|C|} |S_{B_1,R,j}|^2 \right)}_{\text{split quality}} - \left(\frac{1}{|S_{B_2,L}|} \sum_{j=1}^{|C|} |S_{B_2,L,j}|^2 + \frac{1}{|S_{B_2,R}|} \sum_{j=1}^{|C|} |S_{B_2,R,j}|^2 \right) \right]. \quad (10)$$

To search for the best and second-best attributes, we only need to compute the *split quality* denoted in Equation (10) for each split point, instead of the full term of gini impurity reduction in Equation (9). After that, the whole term of Equation (10) is computed for Hoeffding bound judgment. This noticeably simplifies the calculation for each split point.

To further optimize the computation, we eliminate the division $\frac{1}{|S|}$ in Equation (10) by pre-storing and looking up the values in memory. The square-sum calculation in the split quality term is realized with a pipelined multiplier-adder tree.

V. EXPERIMENTS

A. Experimental Setup

In the experiments, we put our main focus on online tree learning. The differences in traditional, batch and online tree learning have been studied in prior works [10], [31] and are not elaborated in this paper. We first implement the software version of our proposed algorithm in StreamDM-C++ [13], the state-of-the-art software toolkit supporting the Hoeffding tree. The parameter settings related to the Hoeffding bound are $n_{min} = 200$, $n_{pt} = 10$, $\tau = 0.05$, $\delta = 10^{-3}$ and $\lambda = 0.01$, according to [10], [13] and [28]. The maximum leaf number is 1024, and the maximum tree depth is 15. We use a 32-bit fixed-point data representation with a 30-bit fraction for numeric attributes, after normalizing the data to within the range of [-1,1], if necessary. We evaluate the design with five large datasets: Bank Marketing (Bank), MAGIC Gamma Telescope (Telescope), Australian New South Wales Electricity Market (Electricity), Covertypes and Person Activity (Person) from the UCI machine learning repository [32] and related works [13], [33]. The optimized hardware is designed in Verilog and implemented on the Xilinx VCU1525 platform [34] using SDAccel 2018.2. The datasets are transferred from CPU to off-chip memory (DDR4) on the FPGA platform through PCIe.

B. Tuning the Number of Quantiles

We tune the number of quantiles in a wide range to evaluate the model performance. The evaluation methodology is *Interleaved-test-then-train*: each sample is first passed through testing before it is applied for training. This is a commonly

TABLE II
RESOURCE UTILIZATION AND FREQUENCY OF FPGA DESIGN.

Dataset	Size	LUT ¹	BRAM ²	DSP ³	Freq. (MHz)
Bank	45211	63079	486	202	308
Telescope	19020	73800	480	184	305
Electricity	45312	54198	384	138	300
Covertypes	581012	169334	1883	1126	170
Person	164860	59401	986	191	266

¹Total No. LUT: 1182240 ²Total No. BRAM: 2160 ³Total No. DSP: 6840

TABLE III
PERFORMANCE COMPARISON: BATCH LEARNING & ONLINE LEARNING.

Method	Platform	Freq.	Exe. time
Batch learning [33]	Intel Stratix IV	200 MHz	118 s
This work	Xilinx Ultrascale+	170 MHz	3.97 ms

used evaluation method for online learning models, and the model performance is evaluated by inference accuracy for the entire datasets. In this way, both the online training and testing phases fully utilize the whole datasets, which is different from offline training methods that require a train-test division and need to separately evaluate training and testing accuracy.

Experimental results in Table I show that the inference accuracy may be degraded significantly as the number of quantiles becomes either too small or too large, especially for the Person dataset. When the quantile number is small, the learning ability of the model may be constrained, because the learned distribution is too coarse-grained to provide effective information. Conversely, if the quantile number becomes too large, the generalization ability may be impaired as well, since the design is more prone to noise in the datasets. Setting the quantile number between 8 and 32 provides high accuracy with desirable robustness. Considering the fact that memory and computation demand is proportional to the number of quantiles, we adopt a unified quantile number of 8 in the hardware design. One can also tune the quantile number to best fit a target dataset. Table II shows the size of datasets and information about FPGA implementation.

C. Comparison with Batch Learning on FPGA

The up-to-date method to cope with decision tree learning with large datasets on FPGA is through batch learning. The work [33] presented a state-of-the-art FPGA architecture for batch-based decision trees. Covertypes is used as the only benchmark in [33], and it serves as the baseline for comparison in Table III. The accuracy and overall resource usage are not given, but study in [10] has proven that both Hoeffding tree and batch tree can lead to the same results for large datasets asymptotically. Table III shows that our proposed online learning design can offer an up to 5-orders-of-magnitude speedup in

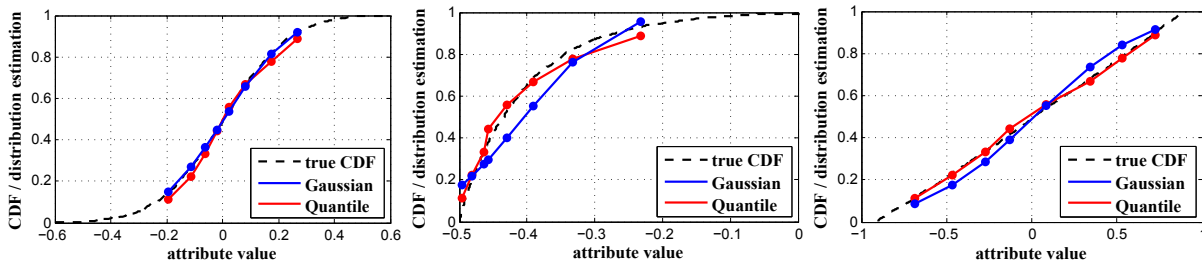


Fig. 10. Gaussian and quantile estimation of true CDFs of three representative attributes from Electricity dataset.

TABLE IV
COMPARISON OF SOFTWARE AND HARDWARE EXECUTION TIME.

Dataset	CPU exe. time		FPGA exe. time	Speedup
	Gaussian	Quantile		
Bank	0.27 s	0.25 s	0.36 ms	750 / 694 ×
Telescope	0.11 s	0.10 s	0.26 ms	423 / 384 ×
Electricity	0.21 s	0.20 s	0.42 ms	500 / 476 ×
Covertypes	6.06 s	6.28 s	3.97 ms	1526 / 1581 ×
Person	0.79 s	0.75 s	0.93 ms	849 / 806 ×

execution time in comparison to [33]. This significant speedup stems from the difference in communication patterns. The work [33] involves a number of rounds of transmission for the same samples from and to the off-chip DDR memory in the training process per batch: it reads the sample set at the start of a split process and writes back the subset of samples in each resulting split. By contrast, our proposed online training architecture only requires reading each sample once in the entire learning process, thus reducing a large amount of high-cost inter-chip communication.

D. Comparison with the State-of-the-art on Processors

StreamDM-C++ [13] reported that Gaussian method provided the best performance amongst prior methods [11]–[14], so it is used as the baseline in this paper. Regarding inference accuracy, our proposed algorithm with eight quantiles outperforms the Gaussian method for all five benchmarks, with 0.05% to 12.3% improvement, as shown in Table I.

The results of CDF approximation using the quantile method and Gaussian method account for this gap in accuracy. Three attributes with representative distributions in the Electricity dataset are selected to illustrate the results, as shown in Fig. 10. The sample set is the subset in the root node before it is split. The CDF of the first attribute is close to the Gaussian function, and thereby, the Gaussian method provides slightly better fitting results than the quantile method. However, regarding the second and third attributes, the quantile method outperforms the Gaussian method. The Gaussian method assumes that the sample distribution conforms with Gaussian distribution, and lead to poor approximation quality for distributions unlike Gaussian. By contrast, the quantile method makes no presumption of any distribution, and hence, it offers accurate approximation for various distributions, including Gaussian distribution. In other words, the quantile method has a wider scope of applicability than the Gaussian method, which accounts for the improvement in accuracy.

For the execution time, we integrate the quantile method in StreamDM-C++ and run this toolkit with both the Gaussian

and quantile methods on the Xeon E5-2680 platform under 2.6 GHz. As shown in Table IV, our proposed hardware designs on FPGA achieve 423× to 1526× speedup over the Gaussian method and 384× to 1581× speedup over the quantile method in software implementation, respectively.

VI. RELATED WORK

Decision tree acceleration on FPGA has been widely investigated. Most of the existing works [35]–[38] have targeted FPGA-based acceleration of inference engines. For decision tree training on FPGA, the work [39] migrated the gini computation from software processing to FPGA implementation. The work [29] sought an SoC solution where the training stage was executed by a soft-core processor on FPGA, while the inference unit was implemented with FPGA logics. The work [40] first designed a complete traditional decision tree training system on FPGA and devised a FIFO-based sorter to facilitate sorting for training, but the memory utilization was high and the maximum size of the datasets was restricted. To reduce memory usage, the work [41] improved upon [40] by employing dataset compression and decompression, with additional data preprocessing time. The work [33] proposed the state-of-the-art of batch learning of decision trees on FPGA. However, it turns out to be inefficient for training large datasets, mainly because it requires transmitting the same samples between the FPGA and off-chip memory multiple times. By contrast, our work introduces a light-weight algorithm along with a hardware-friendly architecture for online decision tree learning, placing no restrictions on the size of datasets and only requiring one-time inter-chip transmission of the datasets. To the best of our knowledge, this paper introduces the first design and optimization of an online decision tree that is applicable to large-scale datasets, and is meanwhile, suitable for FPGA acceleration.

VII. CONCLUSION

Online decision tree algorithms suffer from either high memory usage or high computational intensity with dependency and long latency. In this paper, we introduce an efficient and scalable quantile-based induction algorithm for the Hoeffding tree, and we investigate hardware optimization techniques specific to this algorithm. Finally, we build an online decision tree learning system on FPGA with system-level optimizations. Our design remarkably reduces memory and computational demand, while achieving 0.05% – 12.3% improvement in accuracy and 384× – 1581× speedup in execution time over the state-of-the-art design.

REFERENCES

- [1] J. Chen, H. M. Le, P. Carr, Y. Yue, and J. J. Little, "Learning online smooth predictors for realtime camera planning using recurrent decision trees," in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [2] T. Kaneko, K. Hiramatsu, and K. Kashino, "Generative adversarial image synthesis with decision tree latent controller," in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 6606–6615.
- [3] G. F. Oliveira, L. R. Goncalves, M. Brandalero, A. C. S. Beck, and L. Carro, "Employing classification-based algorithms for general-purpose approximate computing," in *Proc. of ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.
- [4] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 785–794.
- [5] L. Breiman, "Random Forests," *Machine learning*, pp. 5–32, 2001.
- [6] Z.-H. Zhou and J. Feng, "Deep forest: Towards an alternative to deep neural networks," in *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 2017, pp. 3553–3559.
- [7] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.
- [8] L. Breiman, *Classification and Regression Trees*. Routledge, 2017.
- [9] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Elsevier, 2014.
- [10] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proc. of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2000, pp. 71–80.
- [11] J. Gama, R. Fernandes, and R. Rocha, "Decision trees for mining data streams," *Intelligent Data Analysis*, vol. 10, no. 1, pp. 23–45, Jan. 2006.
- [12] G. Hulten and P. Domingos, "VFML – a toolkit for mining high-speed time-changing data streams," 2003. [Online]. Available: <http://www.cs.washington.edu/dm/vfml/>
- [13] A. Bifet, J. Zhang, W. Fan, C. He, J. Zhang, J. Qian, G. Holmes, and B. Pfahringer, "Extremely fast decision tree mining for evolving data streams," in *Proc. of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2017, pp. 1733–1742.
- [14] B. Pfahringer, G. Holmes, and R. Kirkby, "Handling numeric attributes in Hoeffding trees," in *Advances in Knowledge Discovery and Data Mining*, 2008, pp. 296–307.
- [15] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *Proc. of ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD)*, 2001, pp. 97–106.
- [16] E. Ikonomovska, J. Gama, B. Zenko, and S. Džeroski, "Speeding up Hoeffding-based regression trees with options," in *Proc. of International Conference on International Conference on Machine Learning (ICML)*, 2011, pp. 537–544.
- [17] N. Kourtellis, G. D. F. Morales, A. Bifet, and A. Murdopo, "VHT: Vertical hoeffding tree," in *Proc. of IEEE International Conference on Big Data (Big Data)*, 2016, pp. 915–922.
- [18] T. Vasiloudis, F. Beligianni, and G. De Francisci Morales, "BoostVHT: Boosting distributed streaming decision trees," in *Proc. of the ACM on Conference on Information and Knowledge Management (CIKM)*, 2017, pp. 899–908.
- [19] J. P. Barddal, H. M. Gomes, and F. Enembreck, "SFNClassifier: A scale-free social network method to handle concept drift," in *Proc. of Annual ACM Symposium on Applied Computing (SAC)*, 2014, pp. 786–791.
- [20] M. A. Faisal, Z. Aung, J. R. Williams, and A. Sanchez, "Data-stream-based intrusion detection system for advanced metering infrastructure in smart grid: A feasibility study," *IEEE Systems Journal*, vol. 9, no. 1, pp. 31–44, 2015.
- [21] F. Wu, Q. Liu, T. Hao, X. Chen, and Q. Wu, "Online multi-instance multi-label learning for protein function prediction," in *Proc. of IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2016, pp. 780–785.
- [22] Z. Nie, D. Yang, V. Centeno, and K. D. Jones, "A PMU-based voltage security assessment framework using Hoeffding-tree-based learning," in *Proc. of International Conference on Intelligent System Application to Power Systems (ISAP)*, 2017, pp. 1–6.
- [23] L. Rokach and O. Z. Maimon, *Data Mining with Decision Trees: Theory and Applications*. World scientific, 2008, vol. 69.
- [24] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.
- [25] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2001, pp. 58–66.
- [26] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages," *The American Statistician*, pp. 361–365, 1996.
- [27] J. Ho Kim and W. B. Powell, "Quantile optimization for heavy-tailed distribution using asymmetric signum functions," *Princeton University*, 2011.
- [28] A. Althoff and R. Kastner, "An architecture for learning stream distributions with application to RNG testing," in *Proc. of Annual Design Automation Conference (DAC)*, 2017, pp. 15:1–15:6.
- [29] F. Saqib, A. Dutta, J. Plusquellic, P. Ortiz, and M. S. Pattichis, "Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF)," *IEEE Transactions on Computers*, 2013.
- [30] P.-N. Tan et al., *Introduction to Data Mining*. Pearson Education India, 2007.
- [31] Y. Hang and S. Fong, "An experimental comparison of decision trees in traditional data mining and data stream mining," in *Proc. of International Conference on Advanced Information Management and Service (IMS)*, 2010, pp. 442–447.
- [32] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [33] C. Cheng, "Random forest training on reconfigurable hardware," *Ph.D. Dissertation, Imperial College London*, 2015.
- [34] "Virtex ultrascale+ FPGA data sheet," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf
- [35] B. V. Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?" in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 232–239.
- [36] Y. R. Qu and V. K. Prasanna, "Scalable and dynamically updatable lookup engine for decision-trees on FPGA," in *Proc. of High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.
- [37] M. Owaida, H. Zhang, C. Zhang, and G. Alonso, "Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms," in *Proc. of Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.
- [38] D. Tong, Y. R. Qu, and V. K. Prasanna, "Accelerating decision tree based traffic classification on FPGA and multicore platforms," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 11, pp. 3046–3059, 2017.
- [39] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "Interactive presentation: An FPGA implementation of decision tree classification," in *Proc. of Conference on Design, Automation and Test in Europe (DATE)*, 2007, pp. 189–194.
- [40] C. Cheng et al., "Accelerating random forest training process using FPGA," in *Proc. of International Conference on Field programmable Logic and Applications (FPL)*, 2013, pp. 1–7.
- [41] C. Cheng and C. Bouganis, "Memory optimisation for hardware induction of axis-parallel decision tree," in *Proc. of International Conference on ReConfigurable Computing and FPGAs (ReConFig)*, 2014, pp. 1–5.