Adaptive Random Forest on FPGA

Frank Ridder
Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, the Netherlands
f.j.c.ridder@student.utwente.nl

Abstract—**In recent years, machine learning has been increasingly applied to a wide range of real-time applications, with classification tasks playing a critical role in enabling intelligent decision-making. However, the phenomenon of concept drift, where the underlying data distribution changes over time, presents a challenge to maintaining the accuracy of machine learning models. The Adaptive Random Forest (ARF) algorithm addresses this issue by using a combination of multiple Hoeffding Trees and a drift detector to adapt to concept drift. However, as training a forest of trees can be slow, acceleration is needed for real-time use of the algorithm. This work presents the first FPGA implementation of the ARF algorithm, focusing on achieving high hardware efficiency, scalability, and adaptability for different datasets. Our customizable design incorporates various levels of parallelism, resource sharing, and pipelining. Evaluated using four different datasets, the design demonstrates accuracy scores within 0.3% to 10% of GPU ARF implementations. Moreover, our implementation showcases a 1.6x-7x speedup compared to state-of-the-art GPU solutions and a 4x-495x speedup compared to CPU implementations, making it a compelling choice for real-time classification tasks across various applications.**

Index Terms—**Adaptive Random Forest, Field-Programmable Gate Array, Data Streams, Online Learning, Concept Drift, Inference Optimization**

I. Introduction

Machine learning has been increasingly applied in a wide range of real-time applications such as pedestrian detection [1], IP address search [2], energy management systems [3], robotics [4] and network traffic classification [5]. In these applications, classification tasks play a critical role in enabling intelligent decision-making. Machine learning algorithms can analyze large amounts of evolving data streams, allowing the system to adapt and learn from new data in real-time, which is particularly useful for handling dynamic and unpredictable environments. By accurately classifying data into different categories, machine learning can improve the efficiency and accuracy of these applications. For example, pedestrian detection systems can use machine learning to accurately distinguish between pedestrians and other objects, thereby improving pedestrian safety. Similarly, IP address search can benefit from machine learning by identifying patterns in network traffic to detect anomalies or suspicious behavior.

However, machine learning algorithms can be vulnerable to concept drift. Concept drift is a phenomenon that occurs in evolving data streams where the underlying statistical properties of the data change over time, which exists in real-world applications [6]. This can lead to a degradation in the accuracy of machine learning models trained on historical data as they may no longer be representative of the current data distribution. In the case of spam email filtering [7], the distribution of spam emails can change rapidly due to evolving tactics and techniques used by spammers. For example, a spam filter trained on historical data may fail to identify new spam emails that use previously unseen keywords or content patterns. As a result, the accuracy of the spam filter may decrease, leading to more spam emails slipping through to users' inboxes. To mitigate the impact of concept drift, machine learning algorithms must be designed to continuously learn and adapt to new data streams in real-time, updating their models to account for changing statistical properties.

Online learning is a type of machine learning that involves learning from data streams that arrive continuously over time. Compared to offline learning, online learning algorithms update their models incrementally as new data arrives. This approach is especially useful in scenarios where the data is rapidly changing, and the model needs to adapt quickly to new patterns and trends. There are several online learning models available, including Hoeffding Tree [8], K-means [9], and Incremental Support vector machines [10]. Hoeffding Trees use a statistical test called the Hoeffding bound to determine the minimum number of samples needed before extending the tree. Specifically, the Hoeffding bound is used to calculate the minimum number of samples required to ensure that the chosen split is statistically significant with a given confidence level.

The stringent requirements of many real-time applications highlight the need for acceleration in online learning, as it enables the processing of large and rapidly changing data streams. One such example is network traffic classification [11], where an FPGA-accelerated random forest was employed to achieve the necessary throughput for real-time classification of internet traffic in a data center. This example emphasizes the importance of accelerating online learning algorithms in order to meet the demanding needs of real-time applications. By leveraging the parallel processing capabilities of FPGAs, it is possible to significantly improve the performance of online learning models.

While Hoeffding Tree is an online learning model capable of processing data streams quickly, it is not suitable for handling concept drift. To overcome this limitation, the Adaptive Random Forest (ARF) [12] was introduced. ARF uses a combination of multiple Hoeffding Trees and drift detectors like for example the ADaptive WINdowing algorithm (ADWIN)[13] or the Page Hinkley Test[14] to detect and adapt to concept drift. The drift detector signals when the data distribution has changed, and a background tree is trained on the most recent data to replace the affected tree.

The trees in an adaptive random forest are entirely independent of each other, which enables not only parallel processing of the trees but also the possibility of reusing the hardware resources. Furthermore, the independence of the trees allows for the development of deep pipelines by breaking the classification and training process into smaller stages for each tree. This creates an opportunity to accelerate an ARF with dedicated hardware like Graphics Processing Unit (GPU), and Field Programmable Gate Array (FPGA) to make real-time classification possible. This work focuses on accelerating the Adaptive Random Forest algorithm using an FPGA, as they have been shown to offer the highest level of performance and performance per Watt for random forests [15].

Although there have been efforts to accelerate the ARF algorithm using GPUs, the hardware-based acceleration potential of this algorithm has not been fully explored, particularly in the context of FPGAs. Existing state-of-the-art implementations [12][16] are not able to fully exploit the inherent parallelism and adaptability offered by FPGAs, which could lead to significant performance improvements in real-time classification tasks. Our work aims to address this gap by providing a comprehensive FPGA-based solution for ARF acceleration, leveraging the unique advantages of FPGAs in terms of efficiency, scalability, and adaptability to accommodate various datasets and applications.

To the best of our knowledge, this work is the first to implement an ARF on an FPGA with the following contributions:

- We present a customizable design that includes different levels of parallelism, resource sharing, and pipelining to achieve high hardware efficiency, scalability, and the ability to adapt the design for different datasets.
- The design is evaluated using four different datasets and shows similar accuracy scores to a state-of-the-art GPU ARF implementation. Furthermore, we evaluate the performance of the design using these datasets and show a 1.6x, to 7x speedup compared to the GPU implementation.

This thesis is divided into several sections, each providing insights into our FPGA-based solution for the Adaptive Random Forest algorithm. After this introduction, we will give the necessary background in section II, which includes Random Forest, Hoeffding trees, ADWIN, and the Adaptive Random Forest algorithm. This is followed by the framing of our research questions in section III. section IV subsequently presents a discussion of the relevant literature and works related to our research. The architecture of our proposed solution is then outlined in section V, and its detailed implementation, is described in section VI. section VII provides an overview of our experimental setup, preparing us for a comprehensive evaluation of our design presented in section VIII. Lastly, section IX concludes our work and discusses potential areas for future exploration and development.

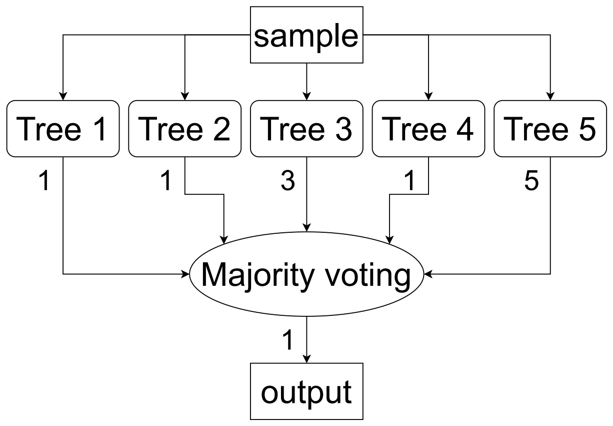


Fig. 1: Example of majority voting

II. Background

The ARF algorithm combines a forest of Hoeffding trees, which are decision tree models designed for processing streaming data, with ADWIN drift detectors, a powerful mechanism for detecting concept drift in evolving data streams. The goal of the ARF algorithm is to maintain high classification accuracy in the presence of changing data distributions by adapting the forest of Hoeffding trees according to the drift detected by the ADWIN drift detectors.

A. Random Forest

Before delving into the details of the Hoeffding trees, it is essential to understand the concept of a random forest and the principle of majority voting, as they form the basis of the Adaptive Random Forest algorithm.

A random forest is an ensemble learning method used for both classification and regression tasks. It operates by constructing multiple decision trees during the training phase and combining their predictions to produce a final output. The core idea behind random forests is that the combined predictions of multiple weak learners (decision trees) often result in a more accurate and robust model. Random forests help in reducing overfitting by training each tree on a subset of the training data otherwise known as bagging (Bootstrap Aggregating).

Majority voting is a technique used in ensemble learning methods to aggregate the predictions of multiple base models. In the case of a random forest, majority voting is applied to the predictions of the individual decision trees within the forest. Each tree makes a prediction for a given input, and the final output is determined by selecting the class that has the most votes from all the trees. A visualization of this can be seen in Figure 1. Majority voting ensures that the overall prediction is more likely to be accurate, as it reduces the impact of individual trees that might have made incorrect predictions due to overfitting or other issues.

B. Hoeffding tree

Hoeffding trees, also known as Very Fast Decision Trees (VFDTs), are a type of decision tree algorithm specifically

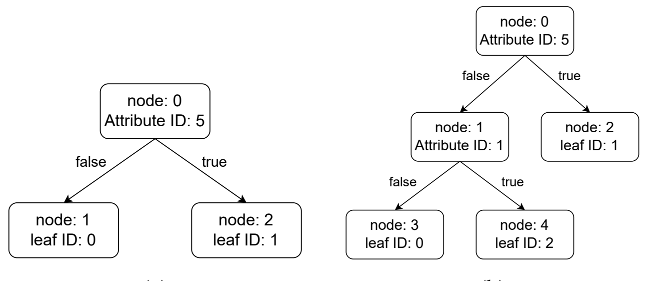


Fig. 2: (a) Hoeffding tree pre-split; (b) Hoeffding tree post-split

designed for data stream classification [8]. The key feature of Hoeffding trees is their ability to make split decisions using the Hoeffding bound, which allows for efficient tree construction with a limited number of data samples. Where splitting is the process of growing the tree by "splitting" a leaf node into an internal node with two new leaf nodes as children. A visualization of this process is given in Figure 2, where node 1, splits into two leaf nodes, node 3 and node 4.

The Hoeffding bound is defined as follows:

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

where ϵ is the bound, R is the range of the random variable, δ is the desired level of confidence (e.g., 0.05), and n is the number of samples processed so far. The Hoeffding bound provides a theoretical guarantee on the difference between the observed mean and the true mean of a random variable with a given probability. Pseudocode for the function responsible for training a single tree is given in Algorithm 1.

Information gain is a metric used in decision tree algorithms to determine the best attribute for splitting a node. It measures the reduction in entropy (uncertainty) when a dataset is partitioned based on a specific attribute. The attribute with the highest information gain is selected as the splitting attribute.

To calculate the information gain for a single attribute A for a given leaf we use the following equation:

$$H(A) = -\sum_{c \in C} p(c) \log_2 p(c) \tag{2}$$

where C is the set of class labels and $p(c)$ is the probability of class c for this attribute in the class distribution as seen in a leaf. In Hoeffding trees, the Hoeffding bound is used in conjunction with the information gain to determine when a node should be split. When the difference in information gain between the best and second-best attribute is greater than or equal to the Hoeffding bound, the node is split using the best attribute. A grace period (line 6, Algorithm 1 is introduced to reduce the amount of split calculation. In addition to using the Hoeffding bound and information gain, the Hoeffding tree algorithm introduces randomness into the tree to further improve performance. This randomness is achieved in two ways: using a Poisson distribution for sample weighting and

Algorithm 1 ARFTreeTrain

```
Input: Samples denoted as $(x, y)$
Input: Tree to train denoted as $t$
Input: Leaf found by tree traversal $l$
1: $k \leftarrow Poisson(\delta)$
2: if $(k > 0)$ then
3:     $UpdateLeafStatistics(l, x, k)$
4:     if $(SamplesSeen(l) \geq GP)$ then // Grace period to
       reduce split calculations
5:         $AttemptSplit(l)$
6:         if $DidSplit(l)$ then
7:             $CreateChildren(l, m)$
8:         end if
9:     end if
10: end if
```

selecting a random subset of attributes during node splitting. The ARF uses the Poisson distribution to assign a weight to each incoming data sample. The weight determines how many times a sample contributes to the updating statistics stored in a leaf node (line 3, Algorithm 1). These statistics keep track of what attribute, what class, and what combination of the class with attributes has been seen when that leaf was traversed to using a given sample. This process stimulates the bagging process in random forests and is known as Online Bagging.

The Hoeffding tree algorithm also adds randomness to the node-splitting process by considering only a random subset of attributes at each leaf node. When a leaf node is evaluated for splitting (line 7, Algorithm 1), the ARF algorithm randomly selects a subset of m attributes from the full set of attributes, where $m = \sqrt{M}$ and M is the total number of attributes. The information gain is calculated for each selected attribute, and the best attribute among the subset is chosen as the splitting attribute. This randomness reduces the change of overfitting for a single tree. However, a single tree is still more prone to overfitting than a forest of diverse trees.

C. Adaptive Windowing

The Adaptive Windowing (ADWIN) drift detector is an important part of the Adaptive Random Forest (ARF) algorithm, providing the ability to detect concept drift and adapt the forest of Hoeffding trees accordingly. Concept drift occurs when the underlying data distribution changes over time, which may cause a decrease in the accuracy of a model trained on previous data. The ADWIN algorithm is designed to detect these changes in distribution by maintaining a sliding window of variable size and monitoring the error rate of the model within this window. An example of this sliding window changing is given in Figure 3 where window W is split into subwindows W_0 and W_1.

The ARF algorithm leverages the ADWIN drift detector at the individual tree level by monitoring the performance of each so-called foreground tree and taking appropriate actions when a warning or drift is detected. To differentiate between a warning and an actual drift, two different delta parameters,

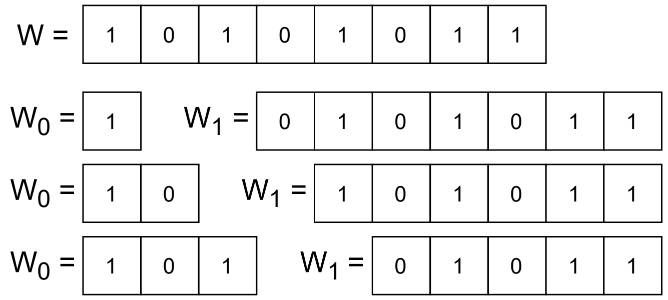


Fig. 3: Example of ADWINs sliding window

$\delta_{warning}$ and δ_{drift}, are used in the ADWIN algorithm. These deltas can be tweaked to make the ARF algorithm react faster or slower to be able to handle various types of concept drift by providing earlier warnings of drift.

Concept drift can be categorized into three main types: abrupt, incremental, and mixed drift. Abrupt drift occurs when the data distribution changes suddenly, while incremental drift represents a gradual change in the distribution over time. Mixed drift is a combination of both abrupt and incremental drifts, often occurring in complex data streams. A visualization of these types can be seen in Figure 4. All of these types of drift can be simulated using generated datasets.

D. Adaptive random forest

Algorithm 2 gives the overall flow of the ARF algorithm. This flow shows how the trees and the ADWIN drift detectors are incorporated into the algorithm. As mentioned before there are foreground trees, these trees are actively used for inference (line 5, Algorithm 2) as their output is used in the majority voting to produce a single output class. The other type of tree is the background tree, these trees are activated whenever the drift detector of a foreground tree gives a warning signal (line 7, Algorithm 2). Then once the drift exceeds the δ_{drift}, the background tree replaces the foreground tree (line 11, Algorithm 2). This way the replacement tree is already partially trained on new data when replacing its foreground tree. By incorporating randomness through Poisson-distributed sample weighting with different weighting for different trees and using random attribute subsets in leaf splitting of the Hoeffding trees, the ARF algorithm creates a forest of diverse Hoeffding trees. This prevents overfitting and therefore creates a more robust model.

Algorithm 2 Adaptive Random Forest

```
Input: Samples denoted as $(x, y)$
Input: Drift threshold denoted as $\delta_d$
Input: Warning threshold denoted as $\delta_w$
Output: Predicted class denoted as $\hat{Y}$
 1: $T \leftarrow CreateTrees(n)$
 2: $\rightarrow B \leftarrow \emptyset$ // Empty set of background trees
 3: for each $(x_t, y_t)$ coming at time $t$ do
 4:     for each $t \in T$ do
 5:         $\hat{y}, l \leftarrow predict(t, x_t)$ // Filter to leaf using tree traversal
 6:         if $C(t, x_t, y_t) >= \delta_w$ then // Drift warning
 7:             $b \leftarrow CreateTree()$
 8:             $B(t) \leftarrow b$
 9:         end if
10:         if $C(t, x_t, y_t) >= \delta_d$ then // Drift detected
11:             $t \leftarrow B(t)$ // Replace tree with background tree
12:         end if
13:         $RFTreeTrain(t, l, x, y)$
14:     end for
15:     for each $b \in B$ do
16:         $\hat{y}, l \leftarrow predict(t, x_t)$ // $\hat{y}$ not used for background trees
17:         $RFTreeTrain(b, l, x_t, y_t)$
18:     end for
19: end for
```

III. Research questions

Given the demand for real-time applications and the need for online learning, it is essential to accelerate machine learning algorithms, such as the Adaptive Random Forest (ARF), to achieve high accuracy in real-time while conserving resources. This research aims to accelerate the ARF algorithm using FPGA technology while maintaining accuracy compared to existing CPU and GPU implementations. To achieve this goal, the study will investigate the parallelization and reuse of the ARF algorithm's components, design an FPGA-based ARF algorithm, and compare its performance to state-of-the-art implementations. The following research questions will guide the study:

- How can the Adaptive Random Forest algorithm be accelerated using FPGA technology while maintaining its accuracy compared to existing CPU and GPU implementations?

The Adaptive Random Forest algorithm is highly modular and many of its components can be reused or parallelized, which presents an opportunity for efficient hardware implementation on an FPGA. For example, each Hoeffding tree requires the calculation of its own Hoeffding bound, and each tree has a drift detector that monitors its performance. A dedicated block that calculates the Hoeffding bound and another one that implements the drift detector could be reused for each tree, significantly reducing the number of hardware resources required. To ensure correct and efficient implementation, the following sub-question needs to be answered:

- What parts of the Adaptive Random Forest algorithm can be parallelized or reused on an FPGA to improve its performance and efficiency?

Not only can parts of the Adaptive Random Forest algorithm be reused, but they can also be configured to have a specific number of trees, nodes, and leaves. This would impact the number of components used by the algorithm on the FPGA. Therefore, it is essential to find a design that fits on the FPGA

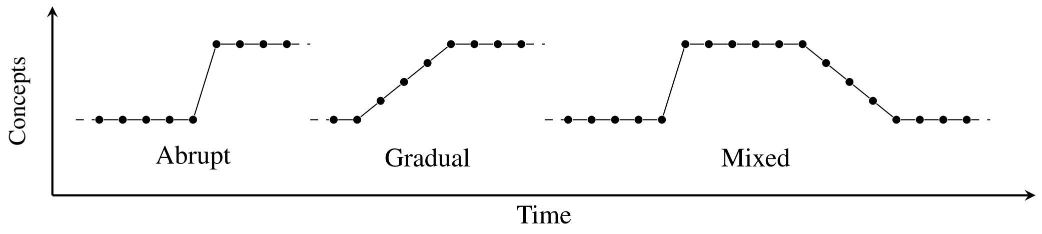


Fig. 4: Visualization of concept drift types

while maintaining a high degree of accuracy. To achieve this, the following sub-question needs to be answered:

- What would an FPGA-based Adaptive Random Forest look like to fit on an FPGA while maintaining a high degree of accuracy?

For classification algorithms, high accuracy is crucial. Therefore, it is crucial that the accelerated Adaptive Random Forest implementation has similar accuracy to the current state-of-the-art implementation of the algorithm. To address this, the following sub-research question is formulated:

- How does an FPGA-based Adaptive Random Forest implementation perform compared to current state-of-the-art implementations?

IV. Related work

In the field of accelerating decision tree algorithms for evolving data stream classification, various implementations have been explored using different hardware architectures. The related works can be grouped based on the hardware platform used.

Decision tree algorithms on CPU Manapragada et al. [17] propose a Hoeffding AnyTime Tree(HATT), this tree builds on the idea of the Hoeffding tree, however surpassing it in accuracy. The HATT however does require additional computational cost over the traditional Hoeffding tree implementation increasing the runtime of the algorithm for some datasets.

Buschjager et al. [18] present optimization for the decision trees in random forests. By exploiting caching behavior, they show they are able to achieve 2x-5x speedup without compromising accuracy. However, this implementation would only be possible for offline learning as the layout of the tree needs to be known for this optimization to work.

Gomes et al. [12] propose two different CPU implementations of the Adaptive Random Forest (ARF) algorithms, a parallel, and a serial version. They show that the parallel implementation is 3 times faster than the serial version without compromising accuracy compared to the serial version. The ARF algorithm can adapt to concept drift, making it suitable for evolving data stream classification.

Decision tree algorithms on GPU Grahn et al [19] present a method for accelerating decision trees and forests by running them entirely on a GPU assigning one thread to every tree allowing many trees to be trained in parallel showing massive speedup compare to other state-of-the-art decision trees implementations. This implementation however lacks the ability to be used in a streaming manner as the decision trees are trained in an offline manner.

Wu et al. [16] focus on accelerating ARF using a GPU and propose a GPU-based State Adaptive Random Forest (GSARF). Their GSARF algorithm stores discarded foreground trees in CPU memory for later use, which enhances accuracy for some datasets. However, where their GPU-based implementation of the ARF algorithm does offer up to 138x speedup compare to the CPU implementation, their design processes batches of incoming data at one time, which would therefore limit performance in real-world data stream classification tasks as the design would need to wait for data to accumulate before being able to process the entire batch slowing the implementation down and creating a large delay between input and output for the first inputs after a batch has been processed.

Decision tree algorithms on FPGA Nakahara et al. [20] present a method for accelerating decision trees on an FPGA by employing Multi-valued Decision Diagrams to reduce the traversal path length, subsequently improving processing speed. However, this technique is based on applying Shannon expansions to already existing trees, making it unsuitable for online learning applications where trees are trained at runtime. Consequently, this approach has limited applicability in evolving data stream classification tasks that require real-time adaptation to concept drift.

Sousa et al. [21] present a high-level synthesis implementation of a Hoeffding tree using FPGA. Their work focuses on achieving efficient acceleration of decision tree algorithms through hardware. Although they demonstrate an 8.3x times speedup compared to a Hoeffding tree implemented on an ARM CPU for the largest dataset for the inference task, their approach does not address concept drift or evolving data streams, which limits its applicability to real-world data stream classification tasks.

Antony et al. [22] propose an FPGA implementation of a Hoeffding tree, which uses adaptive Naive Bayes prediction as a leaf prediction strategy. This approach demonstrates improved accuracy, though it achieves a smaller speedup compared to Lin et al. [23], who present a quantile-based algorithm for Hoeffding tree induction. However, similar to Sousa et al.'s work, both of these approaches do not address the critical issue of concept drift in evolving data stream classification.

In summary, various studies have been conducted to accel-

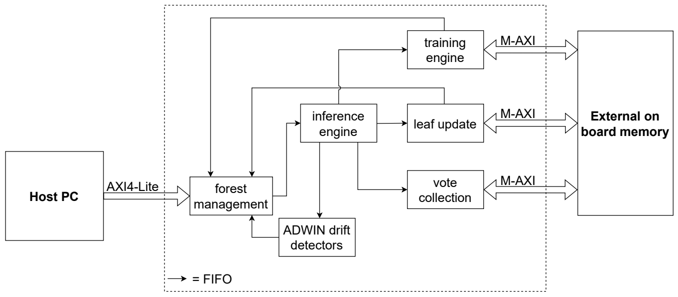


Fig. 5: Overview of proposed architecture

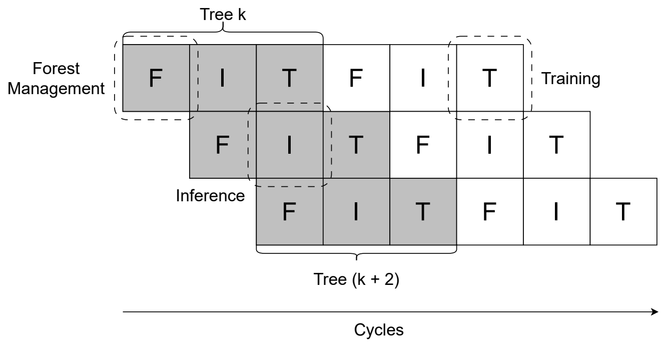


Fig. 6: Three-stage pipeline

erate decision tree algorithms using different architectures and optimization techniques, but none have focused on accelerating the ARF algorithm specifically. Our work aims to address this gap by designing and implementing the ARF algorithm suitable for FPGA acceleration. We seek to maximize speedup while maintaining similar accuracy levels to the state-of-the-art papers accelerating the ARF algorithm.

V. Architecture

A. System Overview

The Adaptive Random Forest (ARF) implementation, as visualized in Figure 5, consists of six interconnected components that work together in a pipelined fashion to efficiently process input data across multiple trees.

The Forest Management unit manages updates from the Training Engine, Leaf Update, and ADWIN Drift Detectors. It also initializes the trees on the first input as there is no previous input for updating the tree. The Inference Engine traverses the trees and extracts the necessary information to forward to the other components for processing.

The Training Engine updates attribute and class statistics, performs splitting calculations, and passes this information back to Forest Management. The Leaf Update component updates the majority class of the leaf found by traversing and passes it back to Forest Management. The ADWIN Drift Detectors are responsible for detecting concept drift and communicating this information back to Forest Management.

The Vote Collection component collects votes from all trees and determines the final classification using majority voting. The order of execution starts with Forest Management, followed by the Inference Engine, and then the Training Engine, Leaf Update, ADWIN Drift Detectors, and Vote Collection components run in parallel. This three-stage pipeline is visualized in Figure 6, where the final four components are grouped under the training stage.

B. Forest Management Unit

Each node in the forest stores an attribute ID, which is used for traversing the tree. Following the approach of the GPU implementation [16], our design is optimized for binary attributes, meaning there is no need to store the value of the attribute used to traverse to the next node. Leaf nodes store their leaf ID instead of the attribute ID. The leaf IDs are then utilized by the Training Engine and Leaf Update components to retrieve additional information about the leaf. The node data, including leaf IDs and attribute IDs, is stored in UltraRAM (URAM). Other data, such as node type and output class of the leaf node, is stored in Block RAM (BRAM). BRAM supports up to two ports. However, more than two components need access to the forest at a time.

As mentioned earlier, when drift is detected by the ADWIN Drift Detectors, it wants to activate background trees and deactivate foreground trees. Additionally, the Training Engine wants to modify the tree structure during node splitting. Since the Inference Engine also requires access to the tree nodes, the Forest Management Unit serves as an arbiter, reducing the number of writes to one and prioritizing changes needed when drift is detected. This approach limits access to the tree nodes to one read and one write per cycle, efficiently utilizing the two ports provided by BRAM and URAM.

C. Inference Engine

The Inference Engine plays a crucial role in the FPGA-based implementation of the Adaptive Random Forest (ARF) algorithm. It is responsible for traversing the Hoeffding trees and extracting the necessary information for classification. To achieve high performance, we employ an m-stage pipeline design in the Inference Engine, where m is the maximum depth of the trees set as a design parameter before synthesis.

This pipelined architecture enables the concurrent processing of multiple trees. In each stage of the pipeline, various operations are performed, including feature comparison, branch selection, and leaf node evaluation. These operations are executed in a highly parallel manner to maximize throughput. Figure 7 illustrates a single stage of the Inference Engine pipeline, showcasing the flow of data and operations done on the data. Where the width of the sample vector is variable and given as a design parameter before synthesis.

D. Training

The Training Engine and Leaf Update component are responsible for updating leaf statistics stored in external onboard memory. Four types of counters represent these statistics. The first counter counts the occurrence of the classes associated with the input samples, the second counts the frequency of

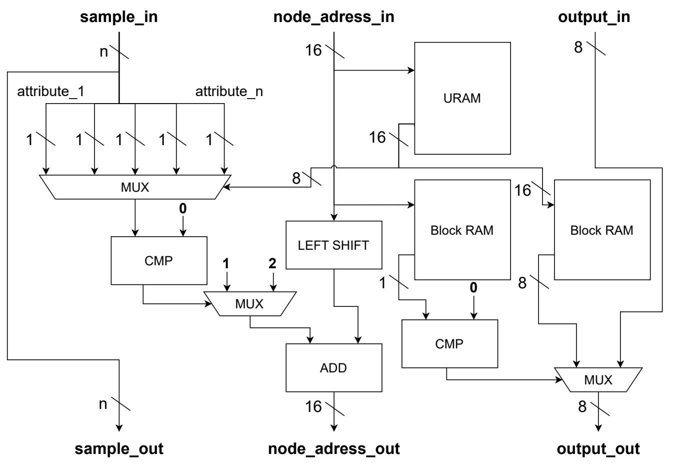


Fig. 7: Single stage of the Inference Engine pipeline

attributes, and the third tracks the occurrences of attributes with the class of the sample. These statistics are updated with a weight generated using a Poisson distribution implemented using a 32-bit Linear Feedback Shift Register (LFSR). Due to the considerable amount of counters, these counters are stored in external onboard memory, as they exceed the capacity of the available on-chip BRAM and URAM.

E. ADWIN

The ADaptive WINdowing (ADWIN) algorithm [13] is a drift detector utilized in the Adaptive Random Forest to identify concept drift in the data streams. ADWIN maintains a sliding window of variable size over the input data and calculates the average of the elements in the window. It compares the averages of two sub-windows to determine if there is a statistically significant difference.

When implementing ADWIN on an FPGA, several modifications are required to optimize its operation for the hardware platform. One of the primary challenges in implementing ADWIN on an FPGA is efficiently managing the sliding window, which consists of a maximum number of buckets and a maximum number of values stored per bucket.

In software implementations, these buckets behave similarly to shift registers. When a bucket is full, the last two values are added to the next bucket, and all other values are shifted two places to be overwritten. However, this approach is not well-suited for FPGA implementations, as it requires frequent access to the BRAM storing the buckets, which could lead to inefficient resource usage and performance bottlenecks.

To address this issue, we employ a technique called virtual shifting when implementing ADWIN on an FPGA. Virtual shifting limits the number of memory accesses to the BRAM by maintaining a counter for each bucket that keeps track of the current position within the bucket. Instead of physically shifting the data within the buckets, virtual shifting updates the counters to virtually shift the data. This approach significantly reduces the number of BRAM accesses, improving the overall efficiency and throughput of the ADWIN implementation on the FPGA.

Furthermore, the software implementation uses two drift detectors, one for the warning and one for full drift detection. However, these detectors use the exact same input data, the only difference is the delta used in the statistical test to detect changes in the subwindows. To reduce storage and runtime, we have opted to only use one drift detector that starts with the $\delta_{warning}$ and uses δ_{drift} for the statistical test once drift was detected.

TABLE I: Dataset dimensions

Dataset	Agrawal	Covertype	KDD99	LED
Attributes	82	44	42	24
Classes	2	7	23	10
Amount of samples	1000000	581012	494020	1000000
Trees used	200	200	120	120
Attribute Subset size	10	7	7	5
Initiation interval	25	25	63	25
Burst read / writes	11	22	63	15

VI. Implementation

The architecture as depicted in Figure 5 is implemented using Xilinx Vitis HLS. Vitis HLS is a high-level synthesis (HLS) tool that allows the conversion of C/C++ code into hardware description languages, such as VHDL or Verilog, suitable for FPGA synthesis. By using Vitis HLS, developers can focus on the algorithm and design aspects, while the tool takes care of generating the hardware implementation and optimizing resource utilization and performance.

The design is synthesized for the Alveo U55C High-Performance Compute Card, which features a Xilinx FPGA with High Bandwidth Memory (HBM) support. HBM allows for significantly higher bandwidth by providing multiple banks that can be accessed by multiple M_AXI ports simultaneously. To exploit this additional bandwidth we store the counters in four different memory banks. Furthermore, the Vitis HLS tool provides the possibility to increase bandwidth by using AXI Burst Transfers for reading/writing chunks of data to or from the HBM in a single request.

Our implementation of the design is able to be tailored to a specific dataset with its respective dimensions. These dimensions are shown in Table I. Furthermore Table I also shows the size of the subset of attributes used for splitting, the number of trees used for the algorithm, the initiation interval of the tree-level pipelined functions, and the amount of burst read / writes made to HBM.

To tailor the design to the dimensions, we provide these dimensions to the design as design parameters before synthesis along with other hyper-parameters of the ARF algorithm like the length of the grace period, the maximum depth of the trees, and the deltas used by the ADWIN drift detectors. This way we are able to synthesize an application-specific implementation as we assume that the dimensions of the data stream will be known during design time.

The iteration interval of the tree-level pipeline corresponds to the time the pipeline needs before it can process the input for the next tree. We tried to target an initiation interval(II) of 25 cycles with a clock speed of 10ns for the tree-level pipelines but because access to external onboard memory is slow the initiation interval for the design tailored for the KDD99 dataset has an interval of 68 cycles because of the large amount of class and attributes.

As some pipelines would be able to run at a faster initiation interval we forced the tool to use the slower initiation interval with the pipeline pragma. This provided the tool with additional cycles for optimizations and remove the need for large FIFO buffers between pipelines to avoid deadlocks.

VII. Experimental setup

We evaluate the performance of ARF on an FPGA using classification accuracy and runtime. The accuracy will be taken on average over the entire dataset. We evaluate the design using four datasets. Two synthetic datasets, Agrawal [24] and LED [25] as well as real datasets, KDD99 Cup [26] and forest covertype [27]. The synthetic datasets have the option to generate different types of drift to test the algorithm. For the real datasets, the entire dataset is used, for the synthetic datasets we use million generated samples as shown in Table I.

All four datasets are also used in Gomes et al. [12], where a CPU implementation is proposed, and Wu et al. [16] where the algorithm is implemented on a GPU to accelerate the algorithm. The results of these works can be used for comparison with our solutions' accuracy and runtime. We evaluated the runtime and accuracy of the CPU and GPU implementation on a Google Cloud VM running Debian 10 with an 8-core processor, 64GB of RAM, and an NVIDIA TESLA V100 GPU.

As mentioned for our design as well as the GPU implementation only accept binary attributes, the datasets mentioned in the previous paragraph do not use binary attributes by default and have to be converted to use binary attributes. For this conversion we use the conversion scripts provided by Wu et al. [16][1]. Wu et al. also provide the scripts used to generate ten different files with varying seeds for every synthetic dataset drift configuration.

As mentioned in the GPU implementation, a buffer of samples is processed in one execution of their kernel, instead of processing a single input at a time, as would be the case in a true streaming implementation. Moreover, they utilize the most frequent class in this buffer as an input to their algorithm and use it as an output for a tree that comprises only a single leaf node. However, in a real streaming implementation, this approach would necessitate preprocessing of the input data. To provide a fair comparison with our design, this feature has been disabled in their implementation for the evaluation performed in this paper.

We get the accuracy of our design by simulating using the simulation tools built into Vitis HLS. To assess the runtime, we consider the following factors: the iteration interval of the tree-level pipelines, the 10ns clock, the average amount of active trees during the simulation, and the number of samples in the dataset. To calculate the performance, we first determine the total time taken to process the entire dataset. We calculate the total time needed to process a given dataset using the following equation:

$$T = II \times C \times T \times S \tag{3}$$

In this equation:

- T represents the execution total time,
- II is the iteration interval of the tree-level pipeline,
- T is the average amount of active trees during simulation,
- C is the clock speed (10 ns), and
- S is the number of samples in the dataset,

Because the reads/writes to external memory are part of the pipeline the latency for actually receiving the data after a request is hidden in the delay to the output of the function-level pipeline. However, this delay is so minimal in relation to the total runtime that it does not show in the total runtime and is therefore omitted from the calculation.

[1]https://github.com/ingako/gsarf

TABLE II: Amount of active trees during simulation

Dataset	Drift Type	#Concepts	Amount of active trees
Agrawal	Abrupt	2	113.81
	Gradual	2	115.94
	Abrupt	3	112.99
	Gradual	3	116.16
	Mixed	3	116.20
LED	Abrupt	2	73.04
	Gradual	2	75.51
	Abrupt	3	73.31
	Gradual	3	74.72
	Mixed	3	74.63
KDD99	unknown	unknown	64.23
Covertype	unknown	unknown	145.81

VIII. Evaluation

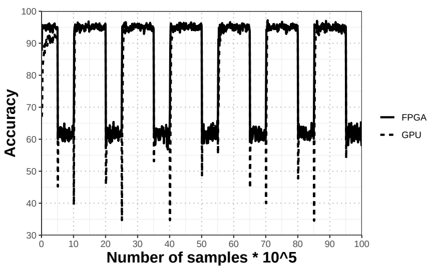
In this section, we evaluate the performance of our Adaptive Random Forest implementation in terms of runtime and accuracy.

As mentioned in the previous section we use the average amount of active trees to calculate the runtime of our implementation. The average number of trees used during simulation is given in Table II. The runtime and accuracy of the three approaches: CPU, GPU, and our HLS-based FPGA implementation are shown in Table III. For the generated datasets, we show the mean as well as the deviation for accuracy.

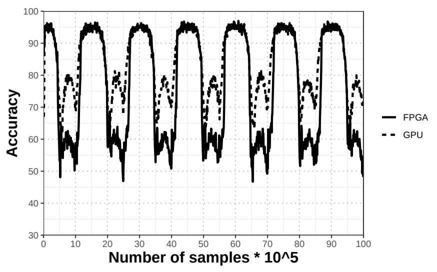
The results obtained reveal that our design achieves comparable accuracy to the GPU implementation, but falls behind the CPU implementation. Our implementation demonstrates better accuracy on the real dataset and the generated dataset with abrupt drift compared to the GPU implementation. However, for generated datasets with gradual drift and mixed drift

TABLE III: Accuracy (%) and Runtime evaluation

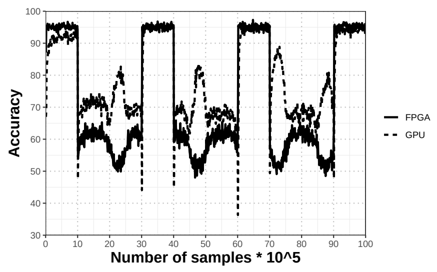
Dataset	Drift Type	#Concepts	Accuracy (%)			Execution time			Speedup: FPGA	
			CPU	GPU	FPGA	CPU	GPU	FPGA	vs CPU	vs GPU
Agrawal	Abrupt	2	92.20 ± 0.11	82.84 ± 0.70	83.13 ± 0.12	1h38m3s	1m58s	28.45s	206.8x	4.1x
	Gradual	2	87.74 ± 0.15	84.10 ± 0.10	76.82 ± 0.07	3h59m9s	3m27s	28.99s	495.0x	7.1x
	Abrupt	3	90.75 ± 0.13	82.09 ± 0.12	78.24 ± 0.15	2h19m21s	1m57s	28.25s	296.0x	4.1x
	Gradual	3	85 ± 0.12	79.02 ± 0.14	69.85 ± 0.13	3h54m45s	3m28s	29.04s	485.0x	7.2x
	Mixed	3	88.95 ± 0.10	79.48 ± 0.12	72.87 ± 0.17	3h45m14s	2m42s	29.05s	465.2x	5.6x
LED	Abrupt	2	72.42 ± 0.20	66.52 ± 0.12	67.78 ± 0.20	15m57s	39.99s	18.26s	52.4x	2.2x
	Gradual	2	65.18 ± 0.14	66.50 ± 0.14	59.36 ± 0.12	17m29s	1m21s	18.88s	55.6x	4.3x
	Abrupt	3	72.58 ± 0.13	67.07 ± 0.25	67.0 ± 0.18	16m16s	42.74s	18.33s	53.2x	2.3x
	Gradual	3	67.51 ± 0.21	66.29 ± 0.12	62.57 ± 0.25	21m54s	1m14s	18.68s	70.3x	4.0x
	Mixed	3	70.45 ± 0.15	67.16 ± 0.10	61.92 ± 0.10	19m58s	58.56s	18.66s	64.2x	3.1x
KDD99	unknown	unknown	99.19	90.81	98.42	1m30s	33.43s	19.99s	4.5x	1.6x
Covertype	unknown	unknown	71.15	57.38	62.15	4m47s	44.34s	21.18s	13.6x	2.1x



(a) Accuracy (%): Abrupt drift

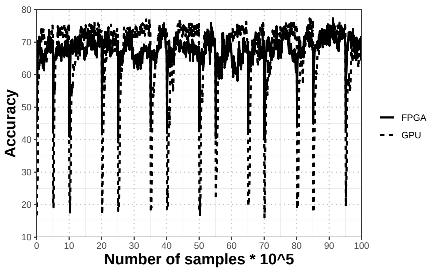


(b) Accuracy (%): Gradual drift

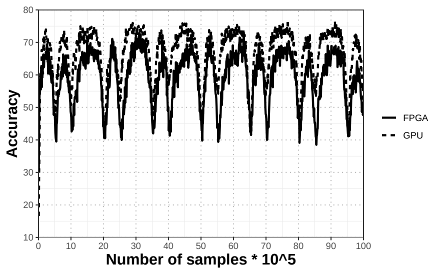


(c) Accuracy (%): Mixed Drift

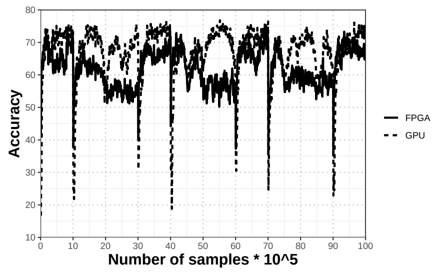
Fig. 8: Accuracy on Agrawal datasets



(a) Accuracy (%): Abrupt drift



(b) Accuracy (%): Gradual drift



(c) Accuracy (%): Mixed Drift

Fig. 9: Accuracy on LED datasets

scenarios, which also incorporate gradual drift, our implementation exhibits lower accuracy. We attribute this to the differences in the split criteria information gain calculation in the different implementations.

Figure 8 shows the response of our implementation as well as the GPU implementations response to the three types of drift seen in the Agrawal datasets. These figures show that our implementation is able to recover faster from drift by having a lower grace period within the Hoeffding tree splitting calculation. The grace period in the GPU implementation is directly related to the size of the batch being processed. Decreasing it could give higher accuracy but as their batch size decreases the runtime will increase. Figure 9 also shows a faster recovery from drift explaining the slight increase in accuracy over the GPU implementation. Furthermore, these figures also give additional insight into the lower accuracy of the datasets containing gradual concept drift. It demonstrates that while the GPU implementation excels in continually adapting to the gradual change in concepts, our implementation swiftly recovers to a comparable level of accuracy once the change is fully realized.

Where our design excels is in the runtime of the algorithm, as demonstrated in Table III. Showing a 1.6x to 7.2x speedup when compared to the GPU implementation where the greatest speedup is achieved with the Agrawal dataset. As Wu et al. [16] explain in their work, their implementation has a

TABLE IV: Resources utilization per dataset

Dataset	Agrawal	Covertype	KDD99	LED
DSP	208(2%)	239(2%)	167(1%)	237(2%)
FF	218536(8%)	263914(10%)	387614(14%)	223125(8%)
LUT	203778(15%)	205108(15%)	299404(22%)	178617(13%)
BRAM	54(1%)	54(1%)	44(1%)	44(1%)
URAM	24(2%)	24(2%)	24(2%)	24(2%)

bottleneck on data transfers from the GPUs to memory to the CPUs memory. Because our design keeps all trees in BRAM on the FPGA we do not experience this bottleneck.

However, for datasets with larger amounts of attributes and classes like KDD99, we experience a different bottleneck in transferring data from external onboard memory to memory on the FPGA. We are able to reduce the impact of this bottleneck by using multiple memory banks that can be accessed simultaneously. As a result, we only have a minor speedup when comparing the runtime of the KDD99 dataset to the GPU and CPU implementation.

Table IV shows the resource utilization of the design tailored for a specific dataset. This table also demonstrates how the dimensions of the dataset impact FPGA resource utilization. As expected, the amount of BRAM increases with the number of trees used in the forest, since most information about the trees is stored in BRAM. Additionally, the table reveals that the amount of Digital Signal Processors (DSPs), Flip-Flops (FFs), and Look-Up Tables (LUTs) grows with the number of attributes and classes in the dataset. The table also highlights that the synthesis tool uses the additional cycles offered by the higher initiation interval of the KDD99 design to reuse DSPs, consequently reducing the number needed for the implementation.

Furthermore, Table IV reveals that the FPGA still has a considerable amount of available space. However, in datasets where data transfer from external onboard memory is not the limiting factor, our ADWIN implementation becomes the bottleneck. Despite the efficiency achieved through virtual shifting, which reduces the number of necessary reads and writes for adding new values, the statistical tests on the shifting window still involve a substantial amount of read and write operations. Because of this, we have an initiation interval of 25 cycles for the ADWIN pipeline. And as we

IX. Conclusion and future work

In this paper, we have presented a novel FPGA-based implementation of the Adaptive Random Forest (ARF) algorithm for addressing concept drift in evolving data stream classification tasks. By utilizing high-level synthesis (HLS) and leveraging various design techniques such as parallelism, resource sharing, and pipelining, our solution achieves a high degree of hardware efficiency, scalability, and adaptability for different datasets.

The experimental evaluation of our design using four diverse datasets confirms that our FPGA-based ARF implementation maintains accuracy levels comparable to CPU and GPU ARF implementations while delivering a significant 1.6x to 7.2x speedup compared to state-of-the-art GPU implementations.

The main research question states "How can the Adaptive Random Forest algorithm be accelerated using FPGA technology while maintaining its accuracy compared to existing CPU and GPU implementations?" In response to this research question, we have successfully demonstrated that our FPGA-based implementation provides an effective solution for accelerating the ARF algorithm while preserving accuracy. By leveraging FPGA capabilities and optimizing the design, we have achieved remarkable speedups without compromising the classification performance compared to other state-of-the-art ARF implementations.

For future work, the independence of the tree can be further exploited by initializing multiple of the same components used in our design apart from the vote collection and using this to reduce the amount of latency in our design. Furthermore, more parallelism could be achieved by more memory partitioning of the counters in external onboard memory, especially for the implementation tailored for the KDD99 dataset.

In future work, this design could be extended to be able to accept both numeric and nominal attributes as well as binary attributes to further increase the usefulness of the design in real-life applications.

References

[1] P. Dollar, C. Wojek, B. Schiele, and P. Perona, "Pedestrian detection: An evaluation of the state of the art," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 4, pp. 743–761, 2012.

[2] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and taxonomy of ip address lookup algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8–23, 2001.

[3] Y. Ji, J. Wang, J. Xu, X. Fang, and H. Zhang, "Real-time energy management of a microgrid using deep reinforcement learning," *Energies*, vol. 12, no. 12, 2019. [Online]. Available: https://www.mdpi.com/1996-1073/12/12/2291

[4] A. Giusti, J. Guzzi, D. C. Cireşan, F.-L. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. D. Caro, D. Scaramuzza, and L. M. Gambardella, "A machine learning approach to visual perception of forest trails for mobile robots," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 661–667, 2016.

[5] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988–2014, 2019.

[6] J. a. Gama, I. Žliobaitundefined, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surv.*, vol. 46, no. 4, mar 2014. [Online]. Available: https://doi.org/10.1145/2523813

[7] P. Lindstrom, S. Delany, and B. Mac Namee, "Handling concept drift in a text data stream constrained by high labelling cost." 01 2010.

[8] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000, pp. 71–80.

[9] P. Domingos and G. Hulten, "A general method for scaling up machine learning algorithms and its application to clustering," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 106–113.

[10] P. Laskov, C. Gehl, S. Krüger, K.-R. Müller, K. P. Bennett, and E. Parrado-Hernández, "Incremental support vector learning: Analysis, implementation and applications." *Journal of machine learning research*, vol. 7, no. 9, 2006.

[11] M. Elnawawy, A. Sagahyroon, and T. Shanableh, "Fpga-based network traffic classification using machine learning," *IEEE Access*, vol. 8, pp. 175 637–175 650, 2020.

[12] H. M. Gomes, A. Bifet, J. Read, J. P. Barddal, F. Enembreck, B. Pfahringer, G. Holmes, and T. Abdessalem, "Adaptive random forests for evolving data stream classification," *Machine Learning*, vol. 106, pp. 1–27, 10 2017.

[13] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," vol. 7, 04 2007.

[14] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, no. 1/2, pp. 100–115, 1954.

[15] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga?" in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 232–239.

[16] O. Wu, Y. Sing Koh, and G. Russello, "Gpu-based state adaptive random forest for evolving data streams," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–8.

[17] C. Manapragada, G. I. Webb, and M. Salehi, "Extremely fast decision tree," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &amp; Data Mining*, ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1953–1962. [Online]. Available: https://doi.org/10.1145/3219819.3220005

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

[19] H. Grahn, N. Lavesson, M. H. Lapajne, and D. Slat, "Cudarf: A cuda-based implementation of random forests," in *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, 2011, pp. 95–101.

[20] H. Nakahara, A. Jinguji, S. Sato, and T. Sasao, "A random forest using a multi-valued decision diagram on an fpga," in *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, 2017, pp. 266–271.

[21] L. Sousa, N. Paulino, J. Ferreira, and J. Bispo, "A flexible hls hoeffding tree implementation for runtime learning on fpga," 12 2021.

[22] A. Antony, D. A, and K. Varghese, "High throughput hardware for hoeffding tree algorithm with adaptive naive bayes predictor," in *2021 6th International Conference for Convergence in Technology (I2CT)*, 2021, pp. 1–6.

[23] Z. Lin, S. Sinha, and W. Zhang, "Towards efficient and scalable acceleration of online decision tree learning on fpga," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 172–180.

[24] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215. Santiago, Chile, 1994, pp. 487–499.

[25] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[26] S. Hettich and S. D. Bay, "The uci kdd archive [http://kdd.ics.uci.edu]," 1999.

[27] J. Blackard, "Covertype," UCI Machine Learning Repository, 1998, DOI: 10.24432/C50K5N.