Accelerating Decision Tree Ensemble Inference with an Automata Representation

Tommy James Tracy II

B.S., University of Virginia (2010)

M.E., University of Virginia (2014)

A Dissertation Presented to the Graduate Faculty

of the University of Virginia in Candidacy for the Degree of

Doctor of Philosophy

Department of Computer Engineering

University of Virginia

August, 2019

Abstract

Decision tree ensembles including Random Forests and Boosted Regression Trees have become ubiquitous in the research domains of medicine, natural sciences, natural language processing, and information retrieval. With increasing data rates and new research into larger ensembles, accelerating the inference rate and reducing the power consumption of this class of machine learning models is critical. It also presents a variety of technical challenges. The random memory access pattern and execution divergence of decision tree traversal results in memory-bound von Neumann implementations.

In this dissertation, we present a series of novel techniques to accelerate decision tree ensembles, by representing their constituent trees as spatial automata that exhibit sequential streaming memory access, and can be executed with high parallelism. We develop novel algorithms and an open source automata framework that allow machine learning and computer architecture researchers to accelerate their applications, as well as stimulate further research into the field of automata-based machine learning. Finally, we present an application study of these techniques and tools with a boosted regression tree-based Learn-to-Rank document ranking model.

Acknowledgments

This dissertation would not have been possible without the mentorship, guidance, and love of many people and organizations.

First and foremost, I would like to thank my family. The PhD was a lengthy and challenging process, and I appreciate the encouragement and love you provided me over the span of the journey. I would like to especially acknowledge my parents **Tom Tracy** and **Christine Tracy** for their lifelong love, encouragement, and for an amazing childhood that built me into the man I am today. I would not have gotten this far without the early push to achieve and the continuous support from both of my excellent parents.

I would also like to acknowledge each of my brothers for always being there for me and providing the brotherly love that can only really be fostered in a loving environment. We grew up together bouncing around the World and were given the freedom to explore and develop into the unique people we are today. Thanks guys. I promise to visit you more frequently now that I'm not a poor graduate student anymore. Thank you: **#2 Benjamin, #3 Ian, #4 Michael, #5 David, and #6 Phillip**.

To my close friends, for all of your support, thank you **Shane Anderson**, **Nathan Brunelle**, **Dan Klopp**, **Jane Plummer**, and **Karolina Sarnowska-Upton**. Being a graduate student was an emotional challenge that each and everyone one of you helped me through it.

To my advisor, **Mircea Stan**, thank you for your guidance and support. You let me explore and define my research journey in ways that I think very few other advisors would have. In some ways this let me broaden my perspective and get hands-on with many different technologies and domains of study. Thank you, Mircea; I truly do think I'm a better researcher, engineer, and life student because of this.

To my mentor **Kevin Skadron**, thank you for your guidance and support. Your hands-on guidance helped me focus my energies towards the end of my dissertation. Also, your approach to leadership, collaboration, and politics was inspiring.

To my mentors of education, thank you **Joanne Dugan**, **Harry Powell**, and **Steven Wilson**. I learned a lot from each of you about what it means to be an inspiring educator.

To **Natalie Edwards**, thank you for all your help and support. You were constantly there to help me with the complications of the UVa bureaucracy and for so much more.

To my collaborators and friends at UVA and elsewhere, including **Kevin Angstadt**, **Chunkun Bo**, **Vinh Dang**, **Patricia Gonzalez**, **Xinfei Guo**, **Mehdi Kabir**, **Benjamin Melton**, **Sergiu Mosanu**, **Mateja Putic**, **Reza Rahimi**, **Alec Roelke**, **Indranil Roy**, **Elaheh Sadredini**, **Jack Wadden**, **Ke Wang**, **Ted Xie**, and everyone else in the **High Performance Low Power** lab, thank you for your support and friendship. I'm looking forward to continuing a lifelong friendship and continue collaboration with you all.

To the National Science Foundation (NSF) and Semiconductor Research Corporation (SRC), thank you for providing the resources to conduct this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning (ML) algorithms make decisions about inputs without being provided explicit instructions on how to get to the result. These algorithms devise the method by which they come to a result by learning from experience. There are many different ML algorithms, called models, that fulfill different use cases. A few examples include *linear regression models*, where a linear model is assumed between input and output values; *decision trees*, where a statistical model is constructed based on example data; and *neural networks*, where a series of nodes are arranged in layers and connected by weighted sums.

Decision trees serve as base learners for a variety of machine learning models including Random Forests [12] and boosted regression trees [30]. These decision tree-based models are versatile with high performance across a significant range of application domains including medicine [87, 60, 28, 118], computer vision [6], information retrieval [112], and in the natural sciences [83, 85, 17]. They also do not require significant optimization effort with only a few hyper-parameters to tune, and are often chosen as a first model for prototyping and testing by researchers and engineers.

With the advent of Internet of Things (IoT) devices and with the rapidly increasing data available to medical [41] and natural science practitioners[57, 69], accelerating the *inference (classification or regression)* rate of these models, as well as increasing their efficiency is of importance. Additionally, contemporary research by Zouh Feng et al. [123, 97], with their introduction of gcForest, showed that variations on very

large cascades of decision tree ensembles could obtain promising accuracy results when compared to deep neural networks. This introduces the potential for large decision tree ensemble models to utilize the research and industry advancements in deep learning. One limitation of the authors' approach is the performance of these large models. The authors highlight in Section 6 of their paper the need for both accelerating their work as well as reducing the memory utilization in order to achieve high throughput, motivating the acceleration and improving the efficiency of decision tree ensemble inference.

Accelerating the inference of decision tree-based models presents a variety of technical challenges. The random memory access pattern of decision tree traversal results in memory-bound von Neumann implementations. Execution divergence while traversing different paths of a plurality of trees prevents efficient parallel execution using SIMD accelerators like GPUs. Although the trees of the ensembles are independently computable, which allows for task-level parallelization on *Multiple Instruction Multiple Data* (MIMD) machines like CPU-based clusters, achieving load balancing and hiding communication overhead remains a challenge. The paths to leaf nodes in the trees are of different depths, resulting in non-uniform execution times, and transferring the *feature vectors* and results between all computing nodes increases communication overhead. In addition, existing decision tree-based machine learning models require buffering and are not conducive to stream processing, a technique whereby data is processed as it arrives.

The Automata Processor (AP) [22] is a non-Von Neumann processor architecture based on the *Multiple Instruction Single Data* (MISD) architectural taxonomy [27]. It can compute thousands of user-defined state machines, called automata, in parallel on a single input data stream. We developed techniques for accelerating decision tree ensembles on the AP as well as other spatial architectures by converting them into spatially-represented automata [92] that can be concurrently executed on an input stream. The random forest algorithm works by comparing the values from a feature vector (representing the input sample) against threshold conditions captured by the root-to-leaf paths in the decision trees. By creating separate automata for all

possible root-to-leaf paths, and by executing these automata in parallel, we were able to achieve significant speedups over the state-of-the-art CPU implementation while also supporting streaming processing.

In order to accomplish the refactoring of these tree data structures, we had to overcome several challenges not addressed by previous automata processing and decision tree model research. Firstly, the classifier feature values, often represented by numerical or statistical values like intensity, TF-IDF, etc. are often expressed as floating point values. Neither floating point values nor floating point operations are natively supported by the automata processing paradigm. We addressed this limitation by developing a *pipelined labeling* technique that represents floating point values with a discrete symbol address space. Pipelined labeling allowed us to maintain fidelity to the original model, reduce the amount of data that the automata needed to process, and also separate and pipeline the thresholding operation from the decision tree traversal. Existing acceleration techniques also reduce the width of feature vectors by converting floating point features into a fixed point representation, albeit at a cost to accuracy[59]; we reduce the need for expensive arithmetic hardware by computing all floating-point comparisons in one step of the pipeline or pushing it off to feature extraction entirely.

Secondly, due to the Multiple Instruction Single Data (MISD) nature of automata computing, all spatially-represented automata consume input feature information encoded in the input stream simultaneously. This requires that all automata be aligned such that they all expect the same feature bytes at the same clock cycles. This deviates from existing decision tree traversal implementations, wherein the order of access to the feature vector's values is determined by the data-dependent tree traversal. We solve this problem by fundamentally restructuring decision tree traversal into the task of filtering an input stream. To do this, we trade off spatial resources for pipelineability, reduce the need for expensive hardware in the decision tree traversal computation, and aligned all decision tree traversals for true parallel and streaming execution.

Finally, in order to reduce the spatial resources required to fit larger decision tree

models onto existing spatial architectures, we develop several compaction techniques called *Automata Folding* techniques. Existing implementation of automata on spatial architectures map state character sets directly. Automata Folding combines character sets into fewer state elements using novel multi-dimensional encoding, reducing the memory requirements of the automata and increasing the spatial efficiency of our representation. This allows us to represent larger models with fewer hardware resources.

We developed two open source research tools, RFAutomata and REAPRpp. RFAutomatat allows researchers to easily convert their existing decision tree ensemble models into automata representations and catalyze automata-based machine learning acceleration. It also allows researchers to explore spatial design tradeoffs and evaluate their models on a variety of automata engines for CPUs, GPUs, Micron's Automata Processor and FPGAs. REAPRpp is a enhanced synthesis tool for converting automata intermediate representations into RTL for FPGAs. This is the final tool required to complete a large set of automata tools for targetting spatial architectures for deployment.

Up until recently, automata engines and architectures were evaluated against sets of regular expressions. Our automata decision tree algorithms are one of several new applications that can be accelerated by automata. In order to better serve the architecture communited, we introduced two new automata benchmarks that include regular expressions and application automata: ANMLZoo and AutomataZoo. We included several automata decision tree models at several key design points into these benchmark suites.

Finally, we accelerate a Learn-to-rank based information retrieval application using this framework on the FPGA as a system-level end-to-end solution achieving significant performance improvements.

1.1 Hypothesis and Contributions

Hypothesis: Decision tree ensemble models can be accelerated on spatial architectures by representing them with finite state automata and optimizing the memory access pattern and model size tradeoffs.

In this dissertation, we demonstrate several novel algorithms and techniques for accelerating decision tree ensembles by representing them as independent automata that run in parallel on spatial architectures. We also describe and demonstrate several tools and benchmarks created for the purpose of furthering the field of automata processing research.

First, we describe a technique for representing decision trees, optimized for von Neumann architectures, as Nondeterministic Finite State Automata (NFAs) on spatial architectures and how to optimize these representations for runtime and spatial resources utilization. We show how these techniques assure deterministic runtimes without modifying the underlying architecture or using approximation methods, and can achieve considerable performance improvements over existing implementations.

Second, we present several additional automata optimization techniques that further improve the resource utilization on spatial architectures. These techniques are applicable across other automata applications, but improve the spatial efficiency of large decision tree-based models on spatial architectures, as well as reducing the length of an encoded input stream representation.

Third, we implement our algorithms in two open source tools: RFAutomata and REAPRpp. RFAutomata is a tool that translates decision tree ensemble models into an automata representation. This tool also provides a slew of features including simulators for various architectures and measurement tools to evaluate the automata representation on CPUs and GPUs. REAPRpp is a synthesis tool for converting this automata representation into hardware description language (HDL) files for deployment on FPGAs. Our tools allow for flexibility by supporting different decision tree ensemble models, automata encoding techniques, and I/O widths. REAPRpp also provides debugging support as well as support for targetting Amazon's AWS F1 in-

stances. We implement a Learn to Rank machine learning application using our tools and techniques. We then evaluate our spatial automata solution deployed on Amazon's cloud-based FPGA instance against a high performing CPU implementation and achieve significant improvements in performance and efficiency.

Our automata models and tools are now part of two automata benchmark suites released to the research community and utilized to evaluate automata architectures and applications. These benchmark suites, namely ANMLZoo[104] and AutomataZoo[106], both serve as the most diverse automata benchmarks currently available.

1.2 Organization

The remaining sections of the dissertation are organized as follows:

Chapter 2: Background introduces decision tree ensembles and automata processing, prior automata acceleration approaches and an introduction to Micron's Automata Processor.

Chapter 3: Decision Tree Automata presents a novel transformation from decision tree ensembles into spatial automata, targeting Micron's Automata Processor.

Chapter 4: Automata Optimizations presents several novel automata optimization techniques for improving the performance of decision tree automata as well as other automata applications.

Chapter 5: Benchmarks and Tools presents several automata research tools and benchmarks used by researchers and in industry.

Chapter 6: Learn To Rank (LTR) as Automata on cloud-resident FPGAs presents a search engine application that we accelerate using our techniques and tools using Amazon's AWS F1 instance.

Chapter 7: Conclusions and Future Work summarizes the dissertation and further discusses the implications and future research directions of our work.

Chapter 2

Background

Machine Learning (ML) as a field of research has been around for decades, but with the explosion of data from Internet of Things (IoT) devices and smart phones, as well as the rise of cloud computing and big data platforms, it has become a critical part of many contemporary computing systems. Whereas in the past complex technologies have remained hidden from consumers and users, “machine learning" is now a discussed issue of our technophillic society and even a feature sought after by the populous for its revolutionary capabilities[78].

ML algorithms make decisions about inputs without being provided a series of explicit instructions to get to the result. They devise the method by which to come to the result, themselves; in essence, they learn their functionality. Machine Learning was formalized by Tom Mitchell as: “A Computer program is said to learn from experience with respect to some class of tasks and performance measure if its performance at tasks, as measured by their performance, improves with experience."[56] These algorithms learn a *mapping function* $f(X) = y$ that maps a given input (X) to an output (y). Some examples of ML algorithms, called models, include Linear Regression, where a linear relationship between an input and the output is learned given a series of example points; decision trees, where a tree data structure is constructed based on statistics of a given set of example data; and neural networks, where a series of weighted edges between layers of nodes are used in an iterative multiply-accumulate pipeline.

2.1 Supervised Machine Learning

Supervised machine learning models are those that learn the mapping function from a set of labelled training data[45]. They are supervised in the sense that they are given large sets of example inputs and outputs with which to compute the mapping function. What differentiates this class of models are the assumptions made by the training algorithms (parameterized vs. non-parameterized), the nature of the outputs (classification vs. regression), and the techniques used to learn and represent the mapping function.

Supervised machine learning models can be broken up into *classification* and *regression* models. A classification model is one that maps an input to a value in a discrete set of classifications. An example classification model, also called a classifier, could determine what of a finite set of objects is depicted in an input image. The classes need to be designated before the machine learning model is trained, and the training data would contain labels only from that set. A regression model is one that maps an input to a continuous, numerical value. An example regression model, also called a regressor, would given an input that represents the current weather pattern, return the future temperature at a given location and time.

2.1.1 Training Machine Learning Models

A supervised machine learning model must first be trained with a labelled training dataset. This dataset is composed of a set of matched input (X) *features* called a feature vector and output (y) *labels*. The goal of the training phase is to learn the mapping function between the inputs and outputs:

$y = f(X)$

What differs between machine learning models is the way this relationship is learned as well as the mathematical model used to represent it. In the case of a linear regression model, the mapping is represented with a linear function. In the case of a decision tree, the mathematical model is represented with a series of comparisons that make up a binary decision tree, where the leaf nodes represent the result.

2.2 Classification and Regression Trees

A *Decision Tree* is a supervised machine learning model in the shape of an inverted tree, with a root node, internal *decision (split)* nodes, and leaf *result* nodes. There are two main types of decision trees: *classification trees* and *regression trees*[11]. A classification tree classifies an input into one of several discrete classes, making it suitable for models with categorical output variables. Regression trees are similar to classification trees, but where the output variable can take on a continuous, numerical value. Decision trees are highly interpretable, whereby the importance of particular features can be calculated by their references in the split nodes of the decision tree. Also, the path taken from the root to the resulting leaf clearly indicates what feature comparisons were made to come to the resulting inference. This makes them particularly useful for medical and science research where interpretability provides valuable research insight[35].

2.2.1 Training Decision Trees

Machine learning models can also be broken up into *parametric* and *non-parametric* models[38]. Parametric models are ones whose mapping functions are based on a predefined mathematical model that requires specific assumptions be made. Non-paramtric models do not make an underlying assumption about the mapping function.

The advantage of parametric models is that training has to simply solve for the parameters of the assumed function, reducing the solution space and speeding up training. They also improve the potential for a training algorithm to converge on the distribution embedded in the training data, assuming the parametric assumptions are valid. The major disadvantage is that if the assumptions are incorrect, the model will poorly fit the training data, resulting in high error.

Non-parametric models, also called distribution-free models, do not make any assumption about the general form of the mapping function, and instead are free to fit any training dataset without a priori knowledge. The advantage of this approach

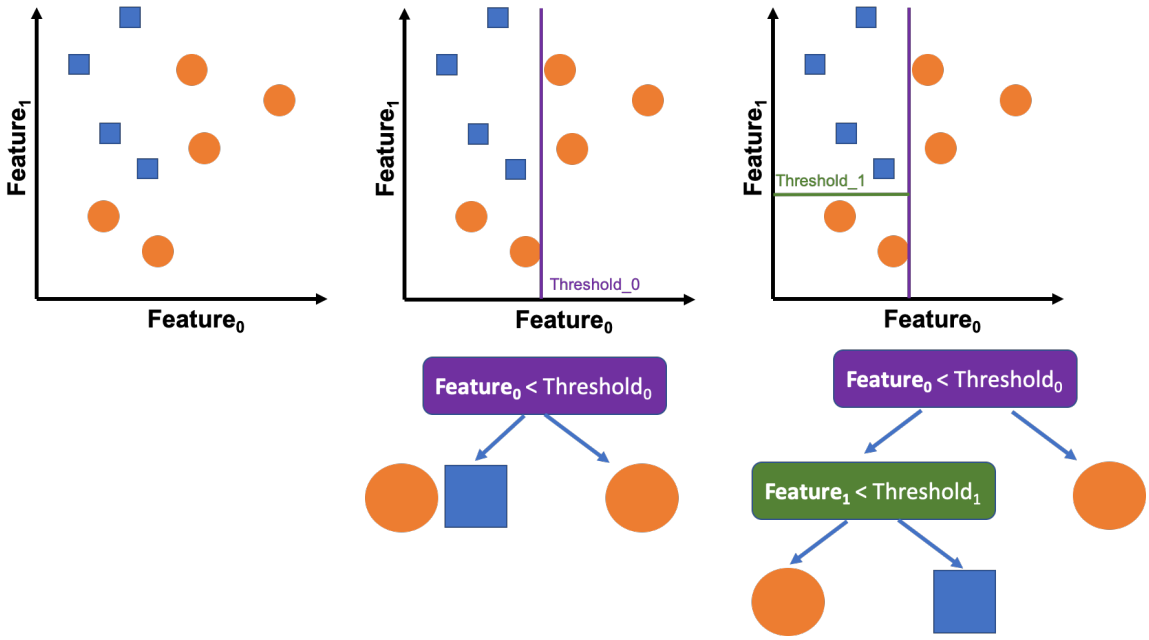


Figure 2-1: Training a decision tree by recursively dividing the training data. [92]

is that no assumptions need to be made about how the mapping function will look. The disadvantage associated with this more general approach is that training becomes more difficult, generally requiring more training data and additional methods to deal with higher variance. Models also tend to grow in complexity with data size.

Decision trees are non-parametric models. Instead, they learn this mapping from a statistical analysis of training data and are "grown" from the root down in an inverted-tree pattern using recursive partitioning of the training data space, each time attempting to split the training dataset between labels.

Figure 2-1 shows a simple example of how this process is done. For our example, we will build a classification decision tree. The left-most plot shows a small training dataset as orange circles and blue squares on a 2-dimensional plot; each of these dimensions represents a feature ($feature_0$ and $feature_1$) of our input space. The purpose of our model is to grow a decision tree to separate these shapes into their own partitions. Once we have the space partitioned such that the shapes are mostly separated into partitioned groups, the model can then assign a classification (orange circle or blue square) to any input features by determining which partition the input features index to.

To do this, the training algorithm finds the feature and associated threshold value to split the training data in order to maximize purity or information gain on each side of the partition. The algorithm tracks a histogram of the different shapes for each partition and determines the ideal split to improve purity in the splits. Our second plot shows the $feature_0$ split at $threshold_0$ that maximizes purity. In this case, the right side of the split is only orange circles (100% pure), but the left side is a mix, so we continue. Next, we select the $feature_1$ split threshold that separates the remaining orange and blue shapes. Every time a split is made, a new node is added to the decision tree. Our first split added the purple node to the decision tree, and the second the green node. In our case, our training data is simple enough to train a decision tree with two split nodes and 100% purity, but this is not usually the case. The training process continues until some purity threshold is met or a maximum depth of growth is met; these restrictions avoid *overfitting* the dataset, where the model extracts more than the intended function, and includes noise in the training data.

Regression tree learning is very similar to classification tree learning, but where the labels of the training data contain continuous values, and the decision tree leaves return continuous, numerical values. Instead of splitting to maximize purity or information gain, regression tree learning maximizes *variance reduction*. Every time a split is made, the variance is calculated for the training data points in each partition. The goal is to reduce the variance, or the stopping criteria S, and effectively partition data points with very similar numerical values. The equation below computes the $StoppingCriteria$ by summing up the square of the differences between predicted and actual label values for each leaf in all trees. This is effectively computing the sum of the variances within all partitions.

$$StoppingCriteria = \sum_{l \in leaves(T)}^{T} \sum_{i \in l} (y_i - m_c)^2$$

Decision Tree training can be concisely summarized as the process of recursively partitioning a training dataset with the intent to maximize a purity metric (classifi-

cation) or minimize variance (regression) while avoiding overfitting. Statistical algorithms are used to determine which feature and thresholds are chosen at each step. The result is a high-dimensional feature space that has been cut up into partitions, where each represents a unique leaf inference value (classification or regression).

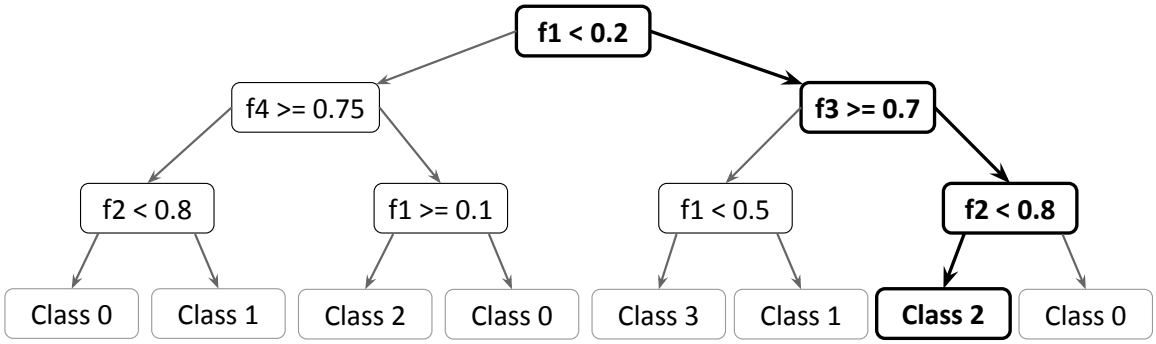
2.2.2 Decision Tree Inference

Decision trees make a decision (classification or regression) from an input that is represented by a vector of numerical feature values called a *feature vector*. These features that make up the feature vector are usually selected and transformed to concisely, and with minimal redundancy, represent the input to the predictive model. A decision for a feature vector is determined by a traversal through the decision tree from the root node through several split nodes, to one of the decision leaves at the bottom. There is a single valid traversal for each input feature vector through the tree, because each split node has two mutually-exclusive outcome paths.

Figures 2-2a and 2-2b show an example of a decision tree and a feature vector. The root-to-leaf path traversed for the given input is shown with bold states. At each split node, the relevant feature is compared against the node's learned threshold value, yielding a left (true) or right (false) traversal. The result for this feature vector is an output prediction taking a continuous, numerical value, in the form of a regression, or a discrete value from a set of classes, in the form of a classification. In this example, the output of our traversal is a classification to $Class$ 2.

2.3 Decision Tree Ensembles

Decision trees can learn any ML data distribution. In the worst case, a decision tree can represent every separate data point in a training dataset by growing deep. This would be accomplished by recursively partitioning the hyper-dimensional space until all partitions have obtained 100% purity, or in the case of regression, 0 variance. Although being able to perfectly represent the training dataset in this way, it would come at the expense of generalizing to all datasets, and *Overfitting* that train-



(a) A decision tree.

(b) feature-vector

Figure 2-2: Classification using a decision tree in the Random Forest.

ing dataset. To evaluate how well the decision tree has learned the training data, validation data is used to evaluate the trained model. Once the decision tree fails to achieve higher accuracy or variance reduction, the decision tree fails to continue learning without tending to overfit.

A decision tree ensemble is a machine learning model composed of a plurality of decision trees. Each individual decision tree comes to its own inference, which are then combined using a function determined by the ensemble technique. Two approaches to using ensembles of decision trees are *Bagging* and *Boosting*.

2.3.1 Bagging vs. Boosting

Bootstrap Aggregation, or Bagging, reduces the variance of decision tree models by training a set of decision trees on different random subsets of the training data. It does this by randomly sampling, with replacement, the training dataset, a process called *Bootstrapping*. A new decision tree is then trained on each of these random subsets. The effect of this approach is that any noise in the training dataset is learned by a subset of the decision trees, and by computing a separate inference value from each decision tree and combining the results with a voting function, the net variance

is reduced.

Boosting is another ensemble algorithm that combines multiple decision trees' results to compute one improved inference. Unlike with bagging, where each constituent model attempts to learn the underlying training data distribution with variance that is offset by voting, boosting trains decision trees incrementally. Each decision tree is small and learns the remaining residuals of the previously learned distribution, incrementally constructing a more effective mapping function. The constituent decision tree inference values are then multiplied by weights and summed together to come to a result.

2.3.2 Random Forest

The Random Forest [12] is a decision tree ensemble ML model that uses bagging for training and a majority voter to combine the inferences from its constituent learners as shown in Figure 2-3. It is called random because not only does it use bagging, where random subsets of the training data are used to train constituent decision trees, but during the training phase a subset of the features are randomly chosen to be considered for the next split. This reduces the tendency for the decision trees to have high coherence, or for them to learn the same mapping function.

Random Forests are ubiquitous across a large group of research domains; some example research includes segmenting rat brains [9], classifying the alcohol dependence of patients based on MRI data[124], meteorological normalization [33], predicting building energy consumption [113], using drones to discriminate weeds from plants [21], and detecting traffic accidents [23]. Random Forests are also being used by the computer architecture research community. Singh et al. proposed Napel[86], a performance and energy estimation tool, where they use a Random Forest model to infer performance and energy usage from micro-architectural parameters and application characteristics. O'Neal et al. propose HLSPredict[64], a tool to predict FPGA performance and power consumption. Random Forests continue to be a popular ML model in many research communities, and it is important to improve their performance and efficiency to further motivate their use.

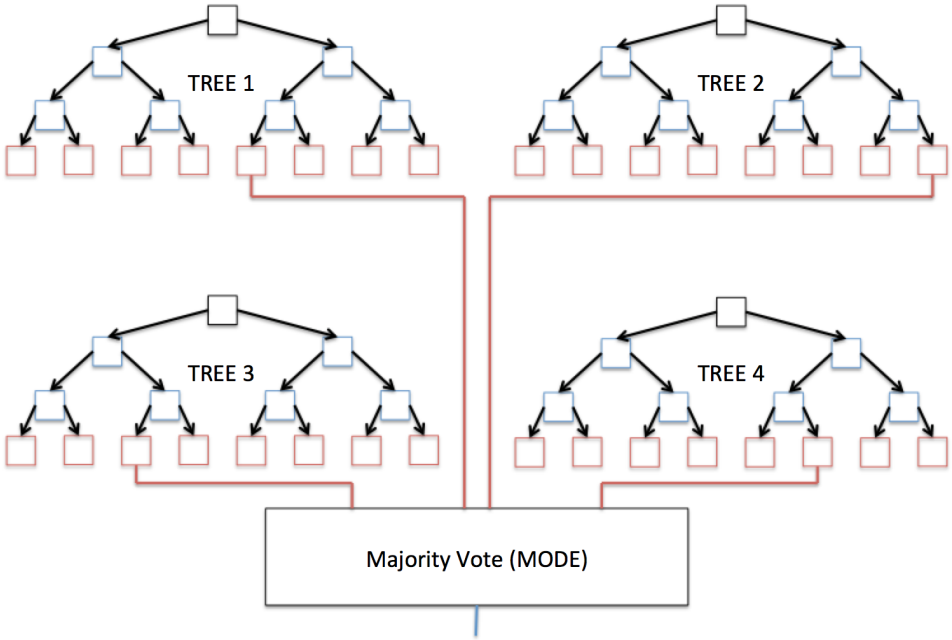


Figure 2-3: The result is the majority vote of the individual classifications.[92]

2.3.3 Boosted Regression Trees

Boosted Regression Trees (BRTs), and the many variations on this approach including AdaBoosted Regression Trees and XGBoosted Regression Trees, are ensembles of regression trees. These models differ from Random Forests in the way that they are trained as well as the way inference is computed. Unlike with Random Forests, BRTs are additive models, where the decision trees are trained on residuals from previously-trained models. Predictions are made by combining tree partial results. Partial results from each decision tree are multiplied by their tree weights and summed for the resulting score:

$$S(x) = \sum_{t=0}^{|T|-1} w_t * l_t(x)$$

Like Random Forests, BRTs are also ubiquitous across a large group of research domains. A small sample of publications from the last two years includes researchers predicting global irradiation forecasts [101], using GIS features to map forests for fire susceptibility [79], and associating microRNAs, or short non-coding RNAs of

approximately 22 nucleotides in length, with various diseases[19]. Another research project uses BRTs to measure the occurance of cyanotoxins in water supplies [61]. The authors also extract Feature Importance information from their trained model to gain insight into what features best indicate the presence of the toxin. It turns out that the pH of the water is the most important feature, followed by the presence of *Microcystis Woronichinia*, a species of freshwater cyanobacteria that can cause algal blooms.

2.4 Why Decision Tree Ensembles?

Although Neural Networks(NN) dominate the contemporary machine learning literature, decision tree ensembles are consistently used across a wide domain of research fields. In particular, research fields where data is pre-processed into tabular, often categorical, data often use decision tree ensembles like Random Forests for classification tasks. They tend to perform well in the general case, and they're very easy to use with only two significant hyper-parameters to tune for in the case of RFs: the number of decision trees in the ensemble and the maximum depth of the decision trees (optional). BRTs have a few more hyper-parameters, but significantly fewer than many other models. Neural Networks, on the other hand, require significant tuning effort, machine learning expertise, and significant training times.

Decision tree ensembles also remain the focus of continuing research in statistics[5, 18] looking at techniques to improve the generalizability of the models with novel training algorithms considering adaptive weighting, dimensional reduction, and variable selection.

2.4.1 Cascaded Decision Tree Ensembles

Additionally, there have been several research projects exploring the use of cascaded decision tree ensembles to represent layered feature processing in a similar way to how neural networks compute inference. These models work by layering decision tree ensembles and passing forward their inference values as input features for the

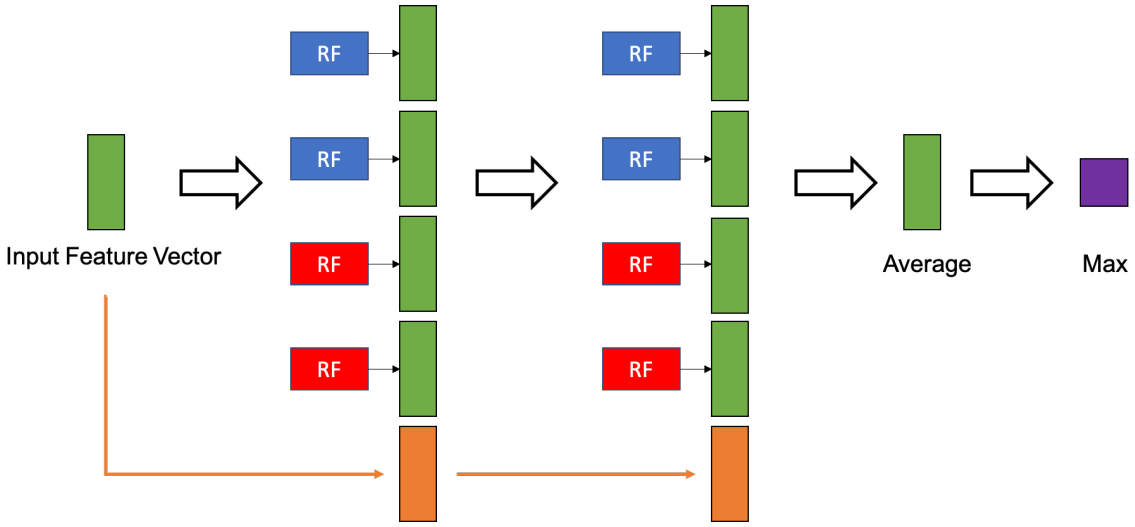


Figure 2-4: The cascaded forest structure proposed by Zouh Feng et al.[123].

next layer. Zouh Feng et al.[123] propose gcForest (multi-Grained Cascade Forest) where they use a combination of Random Forests and Complete Random Forests[49], RFs that are trained with completely random features, to form a new class vector for the next layer to process. These class vectors are then combined with the initial input feature vector and passed forward until the output converges. The authors argue that their approach automatically determines the number of layers necessary and can be trained with far fewer hyper-parameters than a traditional Neural Network. They also demonstrate that they can achieve comparable accuracy to a NN-based solution on the GTZAN[95], IMDB[53], sEMG[82], ORL[80], and MNIST[46] datasets, but that they can do so with less training data.

Figure 2-4 shows the cascaded forest structure of GCForest. The left-most green rectangle represents the input feature vector that is processed by the trained Random Forests (shown in blue) and the trained Completely Random Forests (shown in red). The resulting outputs are interpreted as a vector of class features. These class feature vectors are then combined with the input feature vector (shown in orange), and passed to the next layer. This process is repeated until the MAX of the $AVERAGE$ of the resulting feature vectors converges to the expected output.

Miller et al.[55] with their *Forward Thinking Deep Random Forests* did something very similar. Whereas the gcForest used full Random Forests to generate the class

vector to the next layer, FTDRFs use the output of individual decision trees. This allowed them to achieve the same 98.98% accuracy as GCForest with the MNIST dataseet, but with fewer decision trees. Unfortunately, because the output of all of the decision trees was fed forward, their memory utilization was much higher.

Subsequent work has explored variations on cascaded decision tree ensembles. Ryutaro Tanno et al.[88] merge representation learning from NNs into gcForests by encoding feature representations into edges and leaf nodes. Utkin et al.[96] introduced a weighted approach to gcForest, and Guo et al. introduced BCDForest (Boosting Cascade Deep Forest), to emphasize important features between layers. Kim et al.[43] propose using Random Ferns, or much shallower decision tree ensembles. Other models were also considered; Feng et al.[25] introduced EncoderForest, constructing ensemble-based Autoencoders and Utkin et al.[98] introduced Siamese Deep Forests (SDFs) that implement Siamese Neural Networks[13] as well as implementing weighting of the class vectors between layers.

2.5 Accelerating Decision Tree Ensembles

2.5.1 Accelerating Decision Tree Ensemble Training

There has been a significant amount of work published about accelerating the *training* of decision tree ensembles. Wang et al. introduce DistForest[108], a supercomputer-based parallel RF training framework. Zhao et al. introduced RFAcc[121], a ReRAM based acclerator for training RF models. Zhang et al. with Google introduced a massively parallel decision tree building accelerator for Gradient Boosted Regression Trees with GPUs [120] and Hernandez et al. [36] and Marron et al. [54] introduce two RF training accelerator also with GPUs. Lin et al. [48] propose a decision tree learning approach using FPGAs.

Although important for applications where rapid reconfiguration is important, the training of decision tree ensembles is beyond the scope of this dissertation. Typically, machine learning models are trained offline and then optimized for fast and efficient

inference during runtime. This motivates research energies be spent for accelerating and improving the efficiency of machine learning inference. These are the primary concerns that motivate the novel ideas in this dissertation.

Decision tree ensemble *inference* is bottlenecked by the low-locality of the memory access pattern of the decision tree data structure as well as of the feature vector. Von Neumann architectures with their deep cache hierarchies are heavily optimized for workloads with high temporal locality; the fewer memory accesses made per unit computation, the lower the cache miss rate, and the higher the performance. This ideal workload is very much at odds with the nature of decision tree ensemble inference. Additionally, execution divergence while traversing a plurality of trees of various sizes prevents multi-threaded implementations of Single Instruction Multiple Data (SIMD) accelerators such as general purpose graphics processing units (GPGPUs) from executing tree traversal in parallel.

The mismatch between the ideal workload characteristics of von Neumann processors and the memory access pattern of decision tree ensembles motivates the use of spatial architectures like FPGAs. By partitioning and pipelining the ensemble computations spatially, there is potential for more computations per memory access. Existing implementations explore this tradeoff.

2.5.2 Temporal Architectures

There has been a significant effort made in accelerating decision tree ensembles on CPUs and GPUs as well as spatial architectures including the FPGA. The decision trees that make up these ensemble models are often nonuniform in shape and significant in depth. This prevents the entire ensemble as well as the associated feature vector from residing in the cache memory of modern processors. Additionally, decision tree inference is not compute intensive, but very memory intensive. One threshold comparison is done for each level of the decision tree, requiring the memory architecture load large chunks of the decision tree and feature vector between feature comparisons. CPU approaches have attempted to improve the runtime of the algorithm by maximizing the cache reuse spatially and temporally.

```
i0 = nd[i0].index[(v[nd[i0].fid] >= nd[i0].theta)];
i1 = nd[i1].index[(v[nd[i1].fid] >= nd[i1].theta)];
i2 = nd[i2].index[(v[nd[i2].fid] >= nd[i2].theta)];
i3 = nd[i3].index[(v[nd[i3].fid] >= nd[i3].theta)];

i0 = nd[i0].index[(v[nd[i0].fid] >= nd[i0].theta)];
i1 = nd[i1].index[(v[nd[i1].fid] >= nd[i1].theta)];
i2 = nd[i2].index[(v[nd[i2].fid] >= nd[i2].theta)];
i3 = nd[i3].index[(v[nd[i3].fid] >= nd[i3].theta)];
  ...
```

Figure 2-5: Representing decision trees with structs and indexing with data comparisons from [3]

Researchers have borrowed concepts from modern compiler and database design to improve the performance of decision tree ensemble models on CPUs. For example, Asadi et. al[3] use *predication* and *vectorization* to improve the locality of decision tree traversal and maximize runtime performance. *Predication* is a technique from compiler design research that converts control dependencies into data dependencies, effectively reducing cache misses in exchange for larger models. Branching operations are removed from the execution flow and replaced by trees encoded with struct arrays. Figure 2-5 shows how predication is used. Trees are encoded as a struct array nd, where $nd[i].fid$ represents the feature id of the i^{th} node, and $nd[i].theta$ is the threshold for that node. The nodes are laid out in memory in a breadth-first traversal of the tree, assuming a balanced tree, where node i's left child is located at $2i+1$ and its right child at $2i+2$. *Vectorization* is a technique from database design research that batches decision tree computation to mask cache misses that a traditional sequential approach would incur. This allows temporal architectures to take advantage of pipelining.

Lucchese et. al[51] use an entirely different approach to accelerating ensembles of regression trees by representing traversals with bit vectors. Their algorithm, QuickScorer, uses the commutativity of the AND operation to compute out-of-order tree traversals. We use a similar out-of-order approach, ordering all feature thresholds to be used for simultaneous comparisons, but we pipeline the thresholding, effectively reducing the size of the resulting model. Our implementation also targets spatial

architectures, removing the memory access required by CPU implementations. The authors report the fastest run-times to date by reducing the rates of control hazards and branch mispredictions over the previous state-of-the-art VPRED[3] implementation. Although these approaches have demonstrated considerable improvements over earlier solutions, they are incremental improvements on an algorithm that fundamentally lacks the data locality (both spatial and temporal) necessary for high performance throughput on von Neumann architectures.

2.5.3 Spatial Architectures

There has also been a significant effort made in accelerating decision tree ensembles on spatial architectures like FPGAs. The decision trees can be processed in parallel during inference, making them suitable for pipeline-acceleration on FPGAs. Unfortunately, large ensembles require significant on-chip memory and memory management resources. For this reason, existing FPGA implementations have only been able to accelerate relatively small ensembles on the order of 10 trees with a depth of 12. These existing approaches divide the ensemble into individual trees that are executed in parallel in separate pipelines with threshold comparison units. The heavy memory and buffering costs of these algorithms has resulted in the prevalence of hybrid CPU-FPGA models and often necessitate pruning to ensure that the trees are aligned during processing, allowing for efficient memory utilization. The problem with this approach is that it is still susceptible to the von Neumann bottleneck.

Essen et al.[99] compare multi-core CPU, GPGPU and FPGA Random Forest (RF) implementations and propose accelerating RFs by converting them into a different representation called the Compact Random Forest (CRF) model, pruning the trees and improving the model's pipelinability on FPGAs. Their implementation requires a significant amount of memory management and floating-point comparison hardware that results in their architecture using multiple FPGAs for relatively small RF models.

Owaida et al.[66] recognize the limitations of existing FPGA solutions and presented a hybrid CPU-FPGA approach that can deploy half a million tree nodes in

on-chip memory and achieving a 20x speedup over a 10-threaded CPU baseline. Their approach uses pruning to be able to coerce non-uniform decision trees into a breadth-first memory representation. In fact, the authors cite our work[92] with the Automata Processor mentioning that our implementation cannot handle more than 20 trees of depth 12; we address this limitation in Chapter 4. They continue their work[65] to handle large ensembles by targetting cloud-resident FPGA services from Amazon (AWS) and Microsoft (Catapult).

Nakahara et al. [58] argue that existing FPGA accelerators are based on low-level HDL designs and therefore require longer design time over purely software-based solutions. Their solution is to implement an RF inference accelerator with Altera's SDK and OpenCL. We agree that HDL design is complex and time consuming, but the authors propose a tool flow that generates HDL for the FPGA. Our solution also accepts a Scikit-learn based Random Forest model and generates the HDL, it just does so by constructing HDL instead of using a higher level language in the intermediate. Furthermore, the authors used a fixed point representation to reduce hardware utilization, but at an accuracy cost. This is a common approach to being able to represent decision tree ensembles on spatial architectures without running out of hardware, but our approach does not make this approximation tradeoff.

2.6 Finite State Automata

A Finite State Automaton (FSA)[73] is a mathematical model of computation originating from Computer Science formal language theory that can recognize the class of Regular Languages. It is a recognizer, in that it processes an input stream and returns a binary output $ACCEPT, NOTACCEPT$, depending on whether the machine recognizes the input.

An FSA is a graph of a finite set of node *states* that are connected by directed edges. Each state in the FSA accepts a set of symbols called that state's character set. When an active state matches an input character, the edges that emanate from the matching state activate the next set of states for matching. This process continues

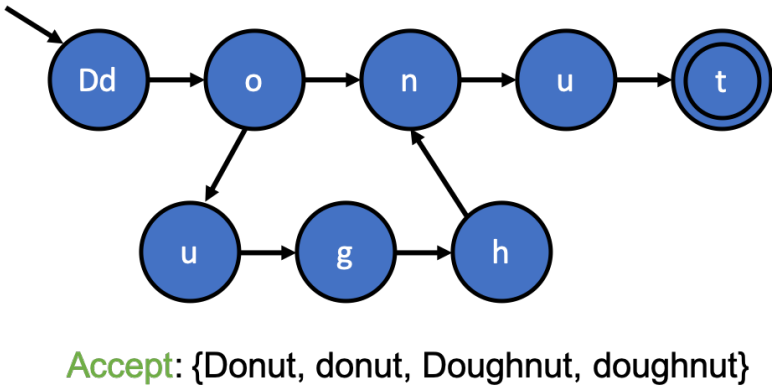


Figure 2-6: A simple FSA that recognizes all valid English spellings of the word "Donut".

until either a *final state*, represented with a double-ring, is reached, or the end of the input stream is reached. If a *final state* is reached, that FSA has found a pattern in the input string and generates an *ACCEPT*. Figure 2-6 shows an example FSA that recognizes all valid English spellings of the word "Donut". The starting state recognizes a 'd' or "D", and each of the two branches of the automaton recognize the subsequent letters of the accepted language $\{Donut, donut, Doughnut, doughnut\}$. Notice how the first two and last three states are shared between the spelling with and without "ugh"; this is an example of prefix and suffix merging, respectively, an optimization to reduce the size of the FSA.

2.6.1 Automata Computing

Automata computing is a computing paradigm whereby one stream of input symbols serves as the input to be processed, and one or more FSAs are run against that input stream. These FSAs are each small pattern matching state machines that look for patterns, effectively computing a Multiple Instruction Single Input (MISD) computation. This model of computing can only be used to describe the class of computing languages called *Regular Languages*, but automata processing has demonstrated performance and efficiency improvements on spatial architectures across a significant application space including big data analysis[10], data mining[110], bioinformatics[75, 89], high energy particle physics[111], machine learning[92], pseudo-random number gen-

eration and simulation[103], and natural language processing[122].

2.7 Processor Architectures

FSAs are graphs represented with nodes that each have their own character sets that are connected with directed edges. This graph represents a language, but not a method by which to process the graph. Efficiently computing FSAs is the focus of on-going architecture research on CPUs, GPUs, FPGAs, and in-memory processing.

2.7.1 von Neumann Architectures

Von Neumann architectures leverage deep cache hierarchies to reduce the cost of memory access by leveraging the locality within workloads. If the workload does not exhibit a high computation-per-load characteristic and has low locality, as in the case with random pointer chasing algorithms, von Neumann architectures will run into the memory bottleneck affectionately known as the "von Neumann Bottleneck." We discussed in a previous section how these architectures map poorly to decision tree ensemble inference. This also holds true to processing finite state automata.

To maximize the performance of finite state automata computing on von Neumann architectures, the FSA graphs are represented compactly, representing transitions between states with look-up tables rather than pointers. To minimize the redundancy of computation, an *active set* of currently active automata states is maintained. For every input symbol only lookups with the tuples generated from the cross product of the active states in the active set and the input symbol need to be evaluated. The lookups yield any subsequent states to then be loaded into the active set. This process is continued until a reporting state is met, in which case a $ACCEPT$ report is logged.

VASim[105] is one such automata engine. For every input symbol, three stages are computed:

1. For each active STE in the active set, check to see if it matches with the input

symbol; if so, buffer it.

2. Enable each child node of the buffered matched states.

3. Special elements (logic and counters) are simulated.

Hyperscan[31] is another automata engine owned by Intel. It uses a series of complex optimizations to maximize performance on CPUs. GPU-based engines[15] have also been implemented to achieve better performance over CPUs. Although they provide a significant increase in thread parallelism, the memory access pattern of automata processing means they do not obtain the same performance advantages[104] that spatial architectures can.

2.7.2 Automata Processing on Spatial Architectures

Many automata applications utilize large numbers of individual automata that all process an input stream in parallel. This Multiple Instruction Single Data (MISD) style of computing maps poorly to von Neuman architectures that assume locality in their application workloads. Several spatial architectures including Micron's Automata Processor (AP) and Field Programmable Gate Arrays (FPGAs) have shown considerable speedups over von Neumann implementations, utilizing the inherent spatial parallelism available to run multiple automata concurrently on one distributed input stream of symbols.

Micron's Automata Processor

Micron's Automata Processor [22](AP) is a reconfigurable, DRAM-based, automata processing architecture. It is a fabric of automata elements, additional logic and counter elements, and an interconnect. The fabric contains *State Transition Elements* (STEs) and boolean elements that can be configured and connected to compute a set of Nondeterministic Finite Automata (NFAs) simultaneously in hardware. NFAs are FSAs where multiple states can be active at once. The AP also contains counter elements that provide additional functionality for more complex machines. A

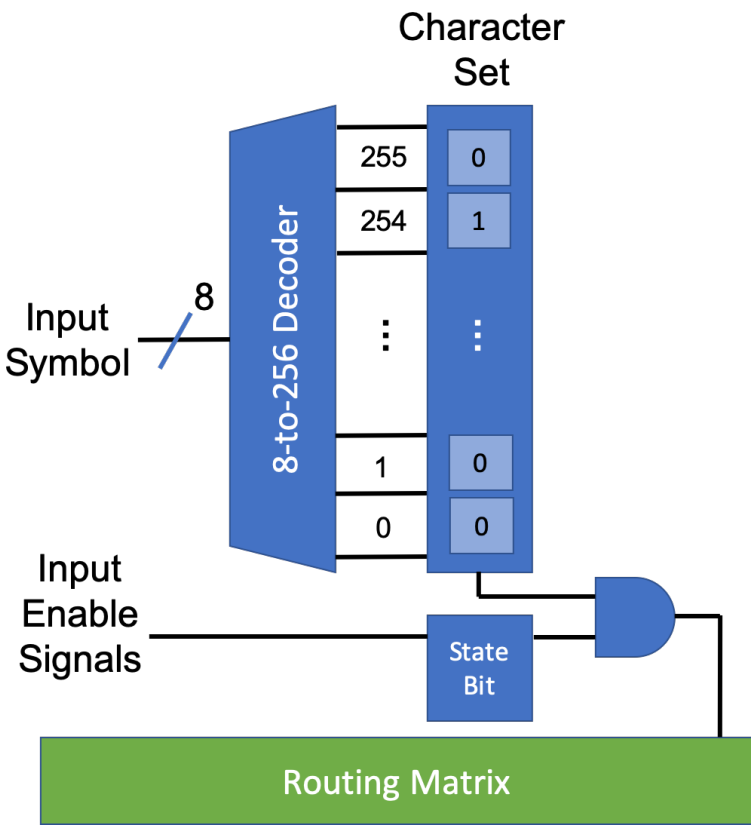


Figure 2-7: An AP State Transition Element (STE). Each STE contains a bit column representing a character set and a state bit that indicates if the state is active or not. An 8-to-256 decoder sets one of the bits in the column high; this value is then AND'd with the state bit to produce the output signal that enables STEs tied to this one's output.

programmer designs automata machines in software, which are compiled and loaded onto the AP.

Figure 2-7 shows a schematic of a single STE in the AP fabric. Each STE contains one 256-bit column of DRAM that contains that state's character set. 256 bits allow the STE to represent all possible combinations of 8-bit values, effectively allowing the STE to represent any subset of 8-bit symbols. In order for a match to be made, that state also needs to be active, so each STE also contains one active bit that is AND'd with the decoded character lookup. If the result of this logical AND is high, all out-going edges are activated and the next states become active for the next input symbol.

NFAs are represented on the AP with STEs and activation edges. STEs represent

states and their corresponding state transition conditions; activation edges describe activation (transition-enabling) relationships between STEs. STEs with incoming edges from the *start state* are marked as *Start STEs*, and STEs with *final states* are marked as *Reporting STEs*. Start STEs can be configured as *start-of-data* STEs which process only the first symbol of the input data stream, or *all-input-start* STEs which process every symbol in the input data stream.

At runtime, all of the automata are loaded onto the processor, and the input data is streamed in as an *input data stream*. This input stream broadcasts one symbol per cycle to all of the AP-chips in an AP *Rank*. On the first clock cycle, only the Start STEs are active which then match the input symbol against the character sets of those STEs. If a match occurs, the matched STE activates all STEs connected to its outgoing connections. This process continues on the next cycle. The counter elements and boolean elements may be used to provide additional logic to these activation signals. If in a cycle one or more reporting STEs are matched, then an output is reported identifying the reporting STE(s) and the offset in the input stream where the match(es) occurred.

A single AP chip contains $49,152$ STEs, 2304 boolean elements and 768 counter elements. An AP board contains 32 such chips, arranged in 4 ranks of 8 chips each. This cumulatively amounts to over 1.57 million STEs, $73,728$ boolean elements and $24,576$ counter elements. All of the chips in one rank can receive a broadcast from a single data stream or can be organized into two *logical cores* of 4 chips each. Each logical core processes the data flow at up to 1 Gbps, allowing a maximum data processing rate of 8 Gbps per board.

Micron's AP hardware is accompanied by a Software Development Kit (SDK) which includes design tools to define, visualize, simulate, compile, load, and run user-defined NFAs on the AP. Using these tools, previous work including biological motif search [75], modeling Markov Chains [103], association rule mining [109], and Brill tagging [122] were developed. Although, these works inspired our research, we report results on actual hardware. In fact, the application for sentiment analysis has been showcased on hardware at the International Supercomputing Conference 2015 (ISC-

15) and the Supercomputing Conference 2015 (SC-15), albeit with restrictions on prototype hardware.

FPGAs

Field Programmable Gate Arrays (FPGAs) are SRAM-based integrated circuits based on a matrix of Configurable Logic Blocks (CLBs) and other hardware resources that can be interconnected with a programmable interconnect. The "Field Programmable" term in the name indicates that FPGAs can be programmed after manufacturing[116]. FPGAs give engineers the ability to rapidly prototype hardware devices.

Xie et al. introduced REAPR[115] (Reconfigurable Engine for Automata PRocessing), a hardware description language (HDL) generator tool for converting finite state automata graphs into a digital circuit, targeting FPGAs. Their LUT-based approach reads the automata intermediate representation, called "ANML" (Automata Network Markup Language), and generates one large flat HDL file that contains statically interconnected STE-like states with one module per state.

REAPR directly translates the finite automata graphs into a digital circuit representation. Xie et al. leverage the one-to-one mapping between characteristics of automata states and circuit elements: the character set can be represented as an 8-input look-up table, the activated status can be stored in a 1-bit flip-flop, and these two elements combined with an AND gate in one STE module.

Each STE module has 4 inputs: 1) an 8-bit global symbol input, 2) a global reset signal, 3) a global clock signal, and 4) a 1-bit signal that determines whether upstream states have been activated. If there are multiple upstream states that activate a particular STE, the outputs of those states are passed through a multiple-input OR gate. The only output from an STE is the AND of whether the STE is activated by other states, and whether the current symbol is in the STE's character set.

REAPR first translates each individual STE's character set into a 256-bit vector, and then creates wires between all of the instantiated states. The connections and

character set bitvectors are hard-coded to enable the FPGA physical implementation tools to maximally apply fine-grained optimizations at various levels, especially logic optimizations. The 256-bit vectors use up precious LUTs and flip-flops in the FPGA fabric, and in case an STE has a very simple character set, then the compiler does not need to use as many resources to implement that logic.

Figure 2-8 shows how each STE is represented by a vector of bits connected to discrete logic components, and how the states are connected by wires. The generated design is a flat dataflow architecture, where symbols enter the automata and cause states to activate downstream states, as shown in Figure 2-9.

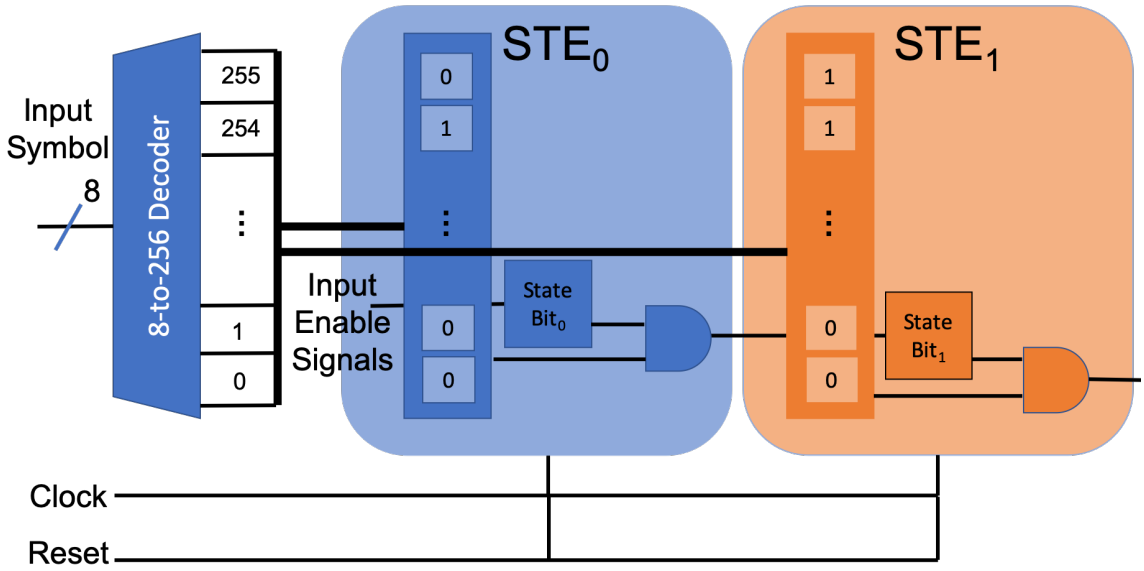


Figure 2-8: All STEs receive the 8-bit input symbol, a clock and reset signal. They are connected in one flat design.

The authors mention that the main disadvantage of their approach is the long compilation times, where the compiler aggressively attempts to minimize logic and increasing compiler effort. Another limitation of REAPR is that the benchmarks used to evaluate it are relatively small, with 2,784 and 100,500 states.

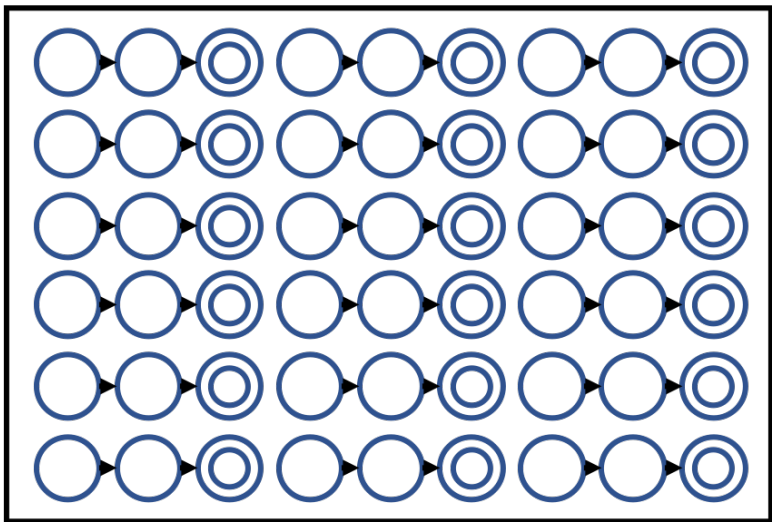


Figure 2-9: All automata are generated in one flat design without any structure.

Chapter 3

Decision Tree Automata

In Section 2.2.1, we discussed how decision trees represent a partitioning of a high-dimensional feature space and that each leaf of the decision tree represents a unique partition. Decision trees are useful for representing this partitioning because they do so in a space-efficient way, and compactness is a primary concern for von Neumann architectures with deep cache hierarchies. Unfortunately, this representation comes at the cost of poor locality in memory access, limiting its inference throughput and efficiency with large models on these architectures.

We replace the decision tree data structures with finite state automata (FSA) [92] we call *decision tree automata (DTA)*, where each separate automaton recognizes a sequence of input symbols that represent features that maps to that automaton's assigned partition. We introduce a series of algorithms and transformations to transform decision trees with their data-dependent memory access patterns to DTA that compute on a sequential stream of data. Our approach also removes the need for significant memory management and buffering hardware that existing spatial architectures use for decision tree inference. Instead, our DTA solution computes on a single, sequential stream of data. With access to large spatial architectures like Micron's Automata Processor (AP) and Field Programmable Gate Arrays (FPGA), we compute all of the FSAs in parallel and achieve significant speedups over a high performance CPU implementation.

3.1 Streaming Automata Inference

The execution pipeline for decision tree automata is shown in Figure 3-1. There are three steps in the inference processing pipeline: labeling the input features, executing the finite state automata against the stream of labels, and reduction, where the resulting automata outputs are combined into one inference value. In the labeling step, continuous features from the input feature vector are converted to their discrete label representation with the help of a lookup into the *Feature Range Lookup Table*. This transformation yields a stream of discrete labels for each input, which are then evaluated in the second step as a stream of symbols with FSAs.

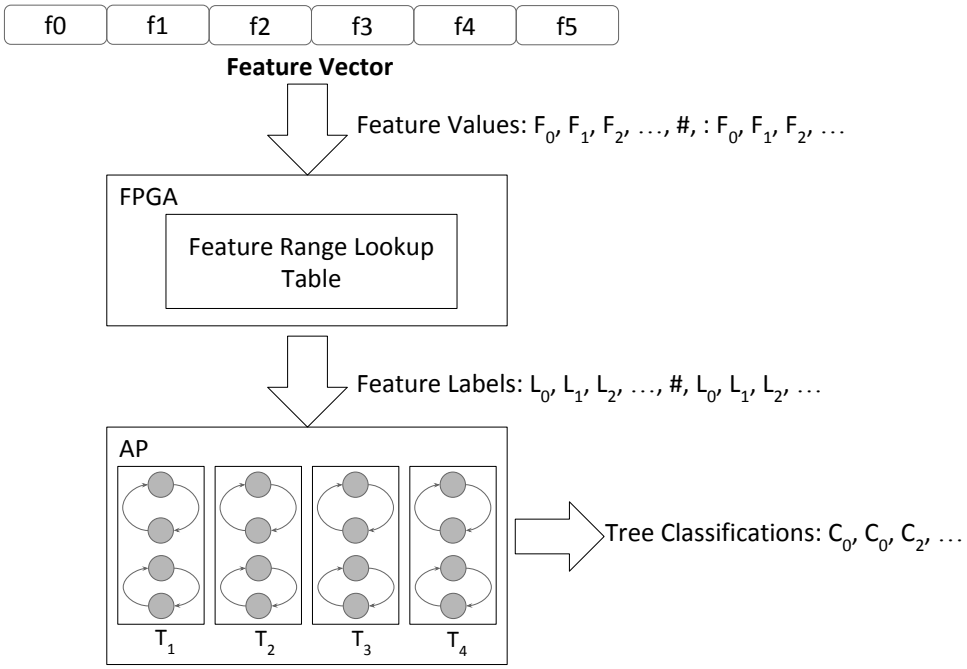


Figure 3-1: The full Decision Tree Automata pipeline. First, the feature vector is converted into feature labels. These labels are then streamed to the Decision Tree Automata on a spatial architecture. [92]

In the second stage, the feature label stream is evaluated against all decision tree automata that have been loaded onto the target spatial architecture. Each automaton looks for a set of labels that correspond to its partition. Because there is a single valid traversal per tree, there will be a single reporting FSA per tree in the ensemble.

Finally, the output reports, one per tree, are sent to the CPU for the reduction

step. In the case of the Random Forest a majority vote is calculated; in the case of boosted regression trees, a multiply and accumulate step is performed. This process is also pipelined and can be computed on the spatial architecture[115] or on the CPU.

3.1.1 Automata Challenges

In order to accomplish this refactoring from tree traversals to sequential stream processing, we had to overcome several challenges not addressed by previous automata processing and decision tree model research. Firstly, feature values like intensity and TF-IDF are often represented by floating-point values. These values are typically incompatible with automata processing, because automata processing uses set membership operations that are oblivious to floating-point encodings. Roy et al.[76] introduce a parallel interval stabbing approach that compares a stream of input bytes against floating-point values, but at a high cost per automaton. Their approach requires several comparisons at various locations in the floating-point byte array. We opt to instead pipeline floating-point comparisons, where we convert floating-point values into fixed-point labels, and generate smaller automata that work on the labels.

We develop a novel *pipelined labeling* technique that discretizes numerical features into discrete symbols. Pipelined labeling allowed us to maintain complete fidelity to the original model, reduce the amount of data that the automata needs to process, and also separate and pipeline the thresholding operation from the decision tree traversal. Existing acceleration techniques reduce the width of feature vectors by converting floating point features into a fixed point representation, albeit at a cost to accuracy[59]; we reduce the need for expensive arithmetic hardware by computing all floating-point comparisons in one step of the pipeline or pushing it off to feature extraction entirely.

Secondly, decision tree traversal is data-dependent, where the index into the feature vector as well as the index of the relevant tree node is not known in advance and is a function of the input. In order to use massively-parallel automata computing, it is necessary to align all FSAs to process the same input feature label at the same clock cycle; doing so would allow us to stream in the feature values once and process all

automata in parallel on that stream. We accomplish this by reshaping our automata such that all automata are aligned to process the same features concurrently. This allows us to stream in our input representation as well as align automata to do stream processing.

We traded off spatial resources for temporal resources. By expanding our model from compact decision trees into many parallel automata, we could reorder our access pattern and enforce perfectly sequential and streaming access of the feature vector. In order to reduce the spatial penality of our method, we developed a compaction technique called *Automata Folding*. Existing implementations of spatial automata map state character sets directly to hardware resources. Automata Folding allows us to combine character sets into fewer state elements, increasing the spatial efficiency of our representation. This allows us to represent larger models with fewer hardware resources.

3.1.2 One Finite State Automaton per Partition

Decision trees can be converted into rules that represent the paths from the root of the decision tree to each leaf node as demonstrated by Quinlan et al. [71]. The authors use this approach to merge decision trees by merging the tree traversal rules. We use a similar approach to convert decision trees into *chains* of operations which we then convert into automata. Figure 3-2 shows how decision trees are converted into parallel chains by breaking the tree shown in Figure 3-3 into separate paths, and generating one chain per path. As shown in Figure 2-1, each of these chains effectively detects if the input feature vector maps to a particular sub-partition.

Each chain represents one partition, or one leaf of the decision tree. The total number of chains represented by all trees (t) in the ensemble of trees (T) is:

$$Automata = Chains = \sum_{t \in T}^{T} leaves(t)$$

Because the number of leaves of a decision tree grows in the exponential of the decision tree's depth, the number of chains also grows in the exponential of the depth.

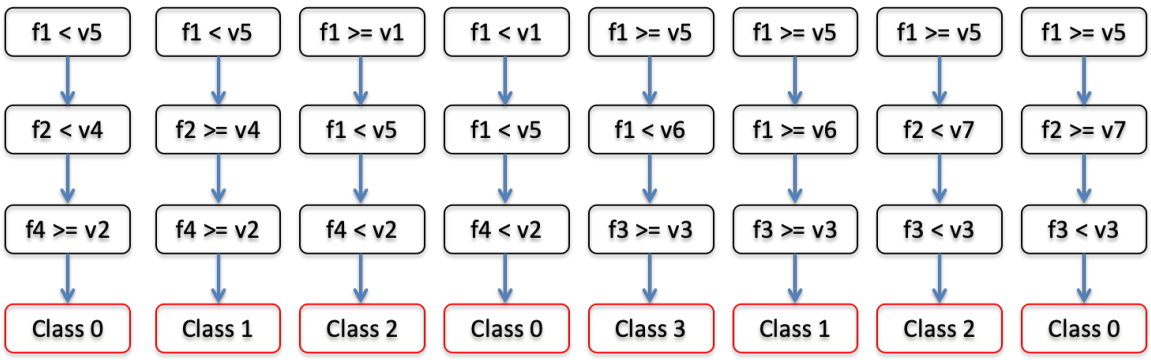


Figure 3-2: Eight decision trees automata that each evaluate one of the paths from *Root* to one of the red *leaf* nodes in Fig 3-3[92]

.

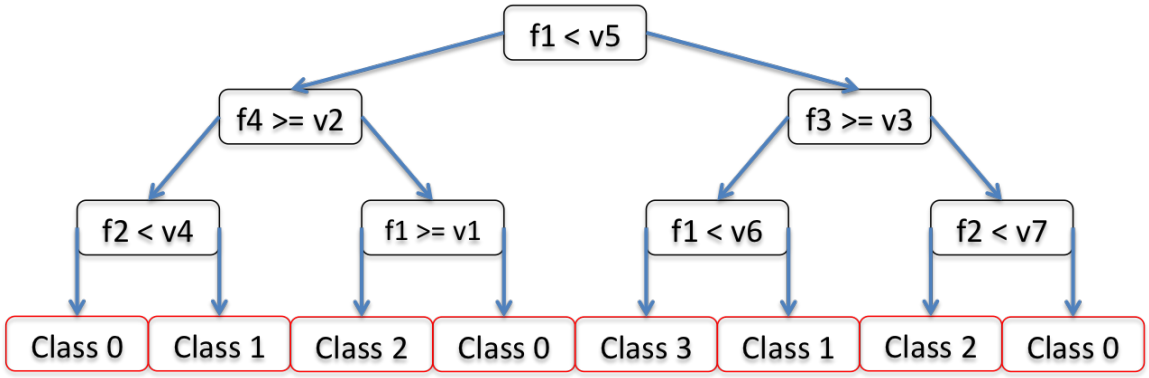


Figure 3-3: A decision tree model. [92]

Our approach scales memory utilization exponentially with decision tree depth and linearly with the number of trees in the ensemble. For this reason, decision tree automata tend to achieve better performance for larger ensembles of shorter decision trees.

Figure 3-4 shows a simple partitioning of the training data space presented in the previous chapter. We replace the decision tree with three small, independently-computable state machines that can all be evaluated in parallel at not only the decision tree granularity, but across the entire ensemble. This means that, with a single sequential stream access of the feature values, an inference can be computed regardless of the decision tree complexity in terms of depth, count or feature permutation.

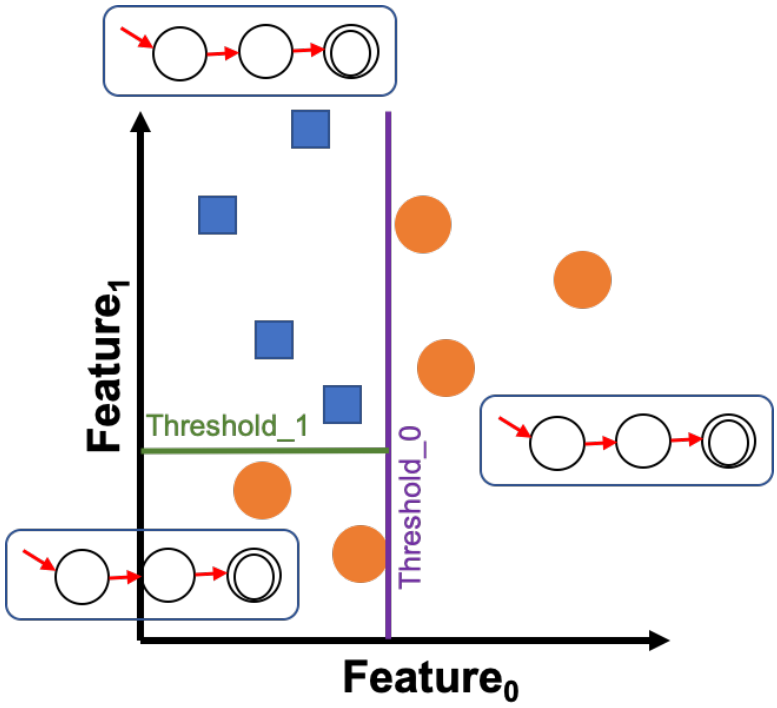


Figure 3-4: Automata can be used to recognize an input string that maps to each of the partitions instead of using decision trees.

.

3.1.3 Aligning Automata

In order to achieve streaming parallelism, all generated chains must be executed simultaneously on the input stream of symbols. Therefore, all chains must share a common feature access pattern. For our decision tree automata, the input stream of feature labels is ordered in a particular way that aligns with the corresponding nodes in each state machine that map to those features at a given input index.

In addition, not all chains consider every feature in the feature vector, so these automata must ignore feature labels that are not considered in the traversal they represent. To accommodate this, we insert 'match-all' states represented with the Kleene Star, *. The order of the nodes for the chains has no effect on the outcome of the computation because each evaluation must be true for the full chain to be true. The commutivity of the chain allows us to reorder the nodes. Figure 3-5 shows how the nodes are rearranged so that at each moment in time (node index from the top) the same feature index is evaluated.

The chains can now be arranged on a spatial architecture and all paths computed in parallel. Each chain acts as a small path detector that evaluates an input stream of features, and will report if that input stream corresponds to the traversal that the detector is looking for. The runtime of this algorithm is linear in the number of features considered by the model (the length of the input stream), and is independent of the depth of the decision trees as well as the number of decision trees as long as the model can fit in the available space. If the model exceeds the available space, the overhead of reconfiguration is significant, motivating the need for more efficient use of spatial resources.

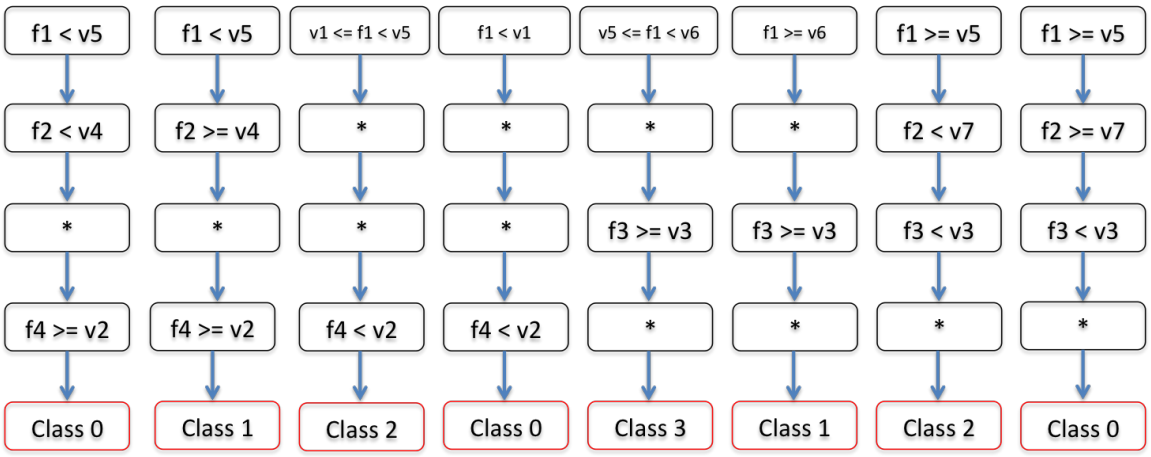


Figure 3-5: The automata after sorting nodes in increasing order and filling unused state STEs with don't care values (*).[92]

3.1.4 Numerical Comparisons to Set Membership

To complete the transformation from decision trees to automata, we convert the continuous input features to discrete symbols as well as to convert the numerical comparison operations that are performed in the chain from threshold comparisons to set membership operations. Conventional implementations of decision tree ensembles compute floating-point comparisons at each inner node. Converting from this continuous approach to one that utilizes discrete features is one approach to increasing the performance of ML classification. By reducing the number of bits used to represent inputs and reducing the size of the model, less hardware is required and

higher clock frequencies can be achieved. This is typically achieved by discretizing the continuous feature space. Lim et al.[47] explained that there are two approaches to *Discretization*:

1. Unsupervised Discretization: Quantize each feature without knowledge of the classes and data. This is typically done by setting an interval width between discrete values or setting equal frequency intervals.

2. Supervised Discretization: Use an ML model to learn discretization intervals based on a training dataset, lumping together feature values that map to the same label.

The authors use decision tree learning to compute the discretization intervals of each feature by breaking a training dataset into partitions (one per leaf) that represent discrete value ranges. They achieved their best performance by using an entropy-based discretization algorithm. Tong et al.[90] accelerate existing FPGA implementations of decision tree ensembles by discretizing feature values in a similar way with their entropy-based *Empirically Optimized Feature Set (EOFS)* approach. We use a discretization approach that perfectly represents the thresholding comparisons, but future work could explore using these other techniques to potentially reduce the amount of ranges required at the expense of accuracy.

We call our approach to discretization *feature labeling* [92]. It differentiates itself from previous approaches by constructing discrete ranges for each feature based on a model trained on continuous data without losing any fidelity to that model. Feature labeling is also an important component of our automata-based approach because by transforming features into discrete symbols, we can perform set-membership operations, the fundamental building block of automata computing.

Our approach works by searching the full ensemble of decision trees and collecting all thresholds evaluated against each feature and binning them. We then sort and list all features' thresholds into address spaces, one per feature. We assigned each range between neighboring threshold values a unique range value; figure 3-6 illustrates this concept. The three plots at the top of the figure show three simple partitions

of the feature space by three different trees. The vertical purples lines are all of the thresholds learned for $feature_0$ and the horizontal green lines are all of the thresholds learned for $feature_1$. We combine all thresholds by stacking these partitions on top of each other to form the plot below. To find the resulting threshold ranges, we project the lines to the axes of the thresholds, and form address ranges for each feature.

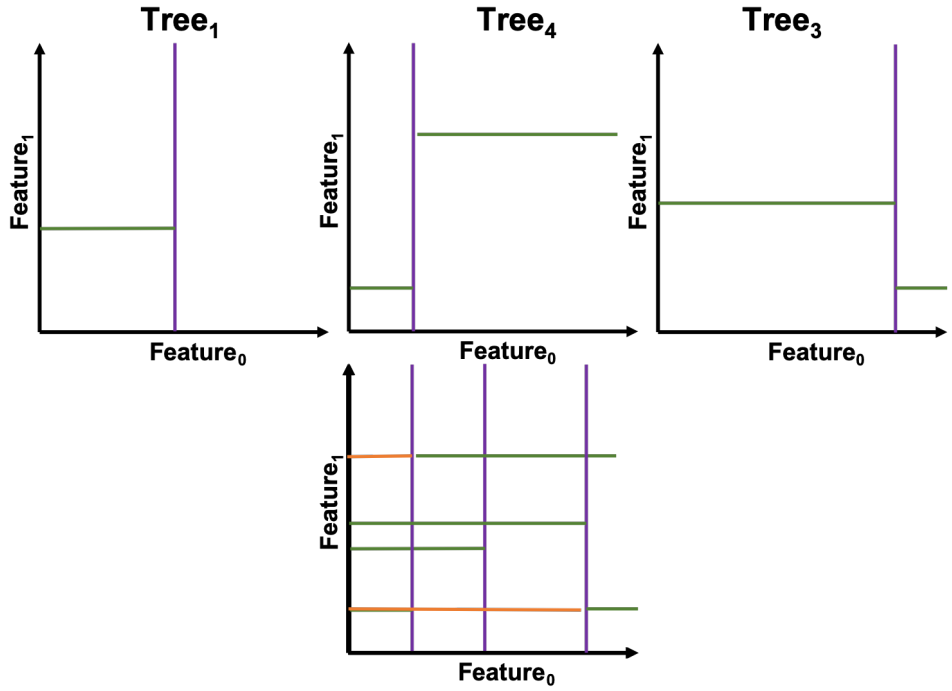


Figure 3-6: Combine all thresholds across all trees into one shared address space.

By representing the ranges between all thresholds with unique identifiers, or labels, it is now possible to perform the same operation as a floating-point comparison by representing the equivalent set of ranges that correspond with the inequalities. The input to the algorithm now corresponds to a range index, and the automaton for each traversal path accepts a set of ranges that correspond to the same mathematical inequality. It is important to note here that we lose no fidelity to the original model.

Figure 3-7 shows an example *Feature Range Lookup Table (FRLT)* with four different features' threshold address spaces: 1, 2, 3, and 4. In this case, for feature 1, there are 3 unique thresholds found in the entire ensemble model: v1, v5 and v6. These three thresholds are sorted and combined, and all ranges between them are labeled with unique range identifiers 0x0 through 0x3. Feature 1 is now represented

with a range index, and the states in the recognizing automata also only accept range indexes, as shown in Figure 3-8. This approach removes the need for floating point comparisons during the inference step of the model, and replaces them with set membership operations. This is a vital step in our transformation, because now we can perform very light-weight pattern recognition on encoded symbols.

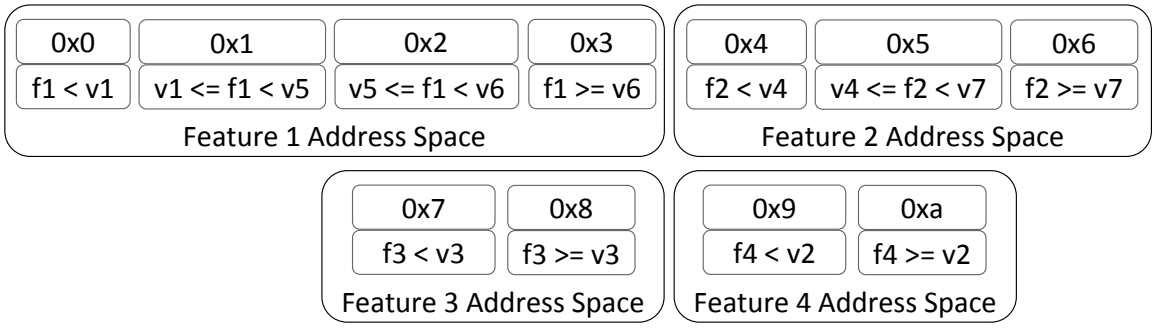


Figure 3-7: The feature address spaces of four different features.[92]

This FRLT is used during the first phase of execution for feature labeling. This can be done on the FPGA or in the feature extraction step. By moving the feature binning step to the feature extraction step, it may be possible to reduce feature extraction power and timing requirements. We leave this for future work.

3.2 Automata Folding

We have discussed converting decision tree traversals into stringy FSAs by breaking the decision trees into chains that represent all tree traversals, one FSA per partition or leaf. We also discussed how we can align all FSAs to consume one feature input stream simultaneously. One limitation of our approach so far is that when mapping our automata to the Automata Processor, one STE is used per feature, per automaton, or in the case of an FPGA implementation, one STE module per feature per automaton. This leads to significant resource requirements, even for moderately sized decision tree ensembles.

We determined that each of these single-feature STEs were being underutilized because of the relatively small number of feature ranges required per feature. By

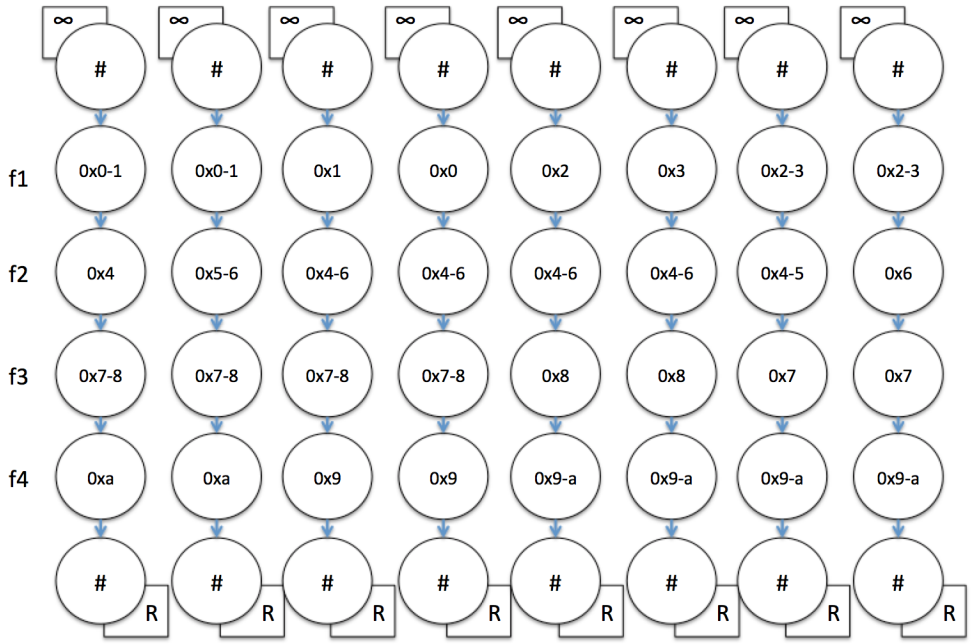


Figure 3-8: Finite State Automata that recognize feature ranges with set membership. Each state only accepts feature range values that correspond with that automaton's traversal path.[92]

combining multiple features into a single state, we could considerably reduce the number of STEs required for each automaton. We use a compaction technique called *Automata Folding*[92] to combine features into fewer STEs by generating loops from our automata chains, effectively folding chains in on themselves.

The features in a Random Forest model typically have differing numbers of intervals (threshold ranges) associated with them. As long as a feature is using significantly less than 255 feature ranges, or the range capacity of an STE, it is possible to represent multiple features in that STE. Automata Folding combines STEs by solving the following optimization problem:

$$\min n : \forall i \in [1, n], \sum_{j=0}^{\lfloor m/n \rfloor} f_{nj+i} \leq C \tag{3.1}$$

where n is the number of STEs used in the automaton, i is the index of the current STE, f_{nj+i} is the number of intervals assigned to feature $nj+i$, m is the total number of features, and C is the capacity of the STE, 255. This optimization function returns

the minimum number of STEs required to represent m features, where the STEs are chained to form a loop. In a simple case where two STEs are required, STE_1 checks feature 1. STE_2 then checks feature 2, STE_1 checks feature 3, STE_2 checks feature 4, and so forth.

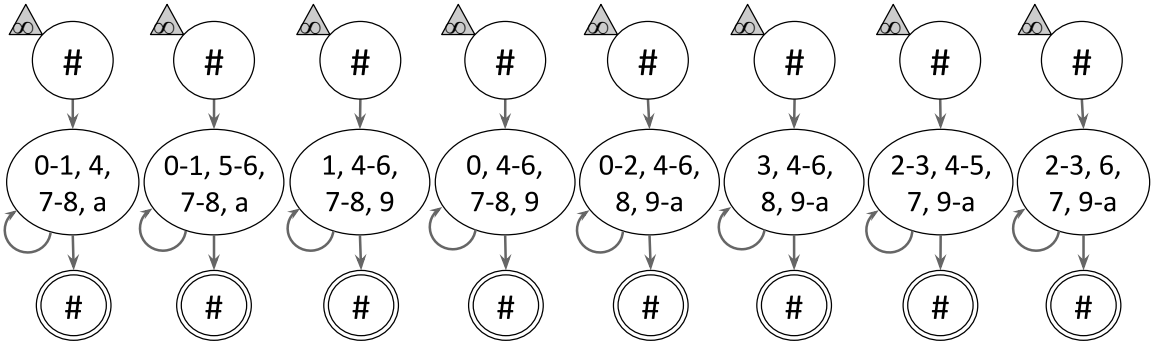


Figure 3-9: Combining features into STEs[92]

Figure 3-9 shows our folded automata. For our simple example, the total number of labels for all of the features is less than 255[1]. Therefore, we need a single STE to check the labels of all of the features. This STE checks the first symbol of the label vector against the possible labels for feature $f1$. If a match occurs, it activates itself to check the second symbol in the label vector against the possible labels for $f2$ and so on. This is possible because the labels for different features are processed on separate clock cycles and the labels assigned to each feature are disjoint in that STE's address space.

If the number of threshold ranges for all features exceeds 255, as is the case with decision tree ensembles of nontrivial size, we extend our loop by adding an additional STE. We then connect the states in a round-robin fashion and use temporal multiplexing to ping-pong between states. The first feature's ranges would map to the first state, the second to the second, the third to the first, and so forth. Solving Equation 3.1 yields the minimum number of STEs required to fit all of the label ranges for all features. It is important to note here that this approach does *not* support features with over 255 feature ranges; we address this in the next chapter.

[1]the symbol space of an STE minus one symbol reserved for the delimiter

3.3 Decision Tree Automata Model

To explore the capacity and performance tradeoffs made by our approach relative to decision tree ensemble model parameters, we explore the number of states required to represent a decision tree ensemble model containing T decision trees with $leaves(t)$ number of leaves per tree (t), with $features$ number of features. We assume that all features are used fewer than 255 times throughout the ensemble; we address the case where features are used more often in the next chapter.

As explained in Section 3.1.2, the number of automata that represent the decision tree ensemble is linear in both the number of trees in the ensemble and the number of leaves per tree. It should be noted that the number of leaves per tree is exponential in the depth of the decision tree, meaning that the spatial resources grow exponential in decision tree depth.

$$Automata = Chains = \sum_{t \in T}^{T} leaves(t)$$

Next, we explore the size of each individual automaton. We showed in Section 3.2 that we could combine logical states with Automata Folding. In the case where all feature ranges could be represented within 255 ranges, the best case scenario would allow us to represent the full automaton with a single state. In the worst case, Automata Folding would not provide any spatial reduction, resulting in the number of states per automaton being equal to the number of features in the input stream.

$$StatesPerAutomaton_{BestCase} = [1, features]$$

Combining these cases, we determine that the total number of spatial states required for our automata decision tree model is:

$$TotalNumberOfStates = StatesPerAutomaton \times Automata$$

With best and worst case scenarios resulting in:

$$TotalNumberOfStates_{BestCase} = \sum_{t \in T}^{T} leaves(t)$$

$$TotalNumberOfStates_{WortCase} = features \times \sum_{t \in T}^{T} leaves(t)$$

Assuming that all automata can be represented on the spatial architecture concurrently, the runtime per inference is then linear in the number of features, as the length of the input feature vector is the number of features. This constant inference time is one benefit of using automata for decision tree traversal.

3.4 Experimental Analysis

We implemented our automata-based design on the Automata Processor as a proof of concept. Although we were able to run our design on AP hardware, because the AP was a prototype, we were unable to achieve high performance results. For this reason, we ran performance simulations with the performance metrics provided by Micron; we provide those in our results section below. We then compared our AP implementation against a high performance CPU implementation using Random Forest models trained on two different datasets: the MNIST handwritten digits database [46] and the Sanders Twitter Sentiment Corpus [81].

3.4.1 Datasets

The MNIST handwritten digits dataset[46] contains labeled images of handwritten digits, where the classification for each image takes on one of the values from 0 to 9. Each input sample is represented by a 28x28 pixel two-dimensional array representing the greyscale values of the image after being centered and scaled. This dataset has become a commonly used dataset for evaluating machine learning algorithms and serves as a good example for our problem because of the relatively high feature count and learning complexity.

The second dataset is the Sanders Twitter Sentiment Corpus [81]. This dataset contains one large set of Twitter messages and their associated sentiments. The positive, negative, neutral and irrelevant sentiment classifications indicate the inferred author's sentiment when posting the tweet.

3.4.2 Training

The two applications that span very different domains were used to train two diverse sets of Random Forest models using version 0.16.11 of the Scikit-Learn [68] machine learning framework. We vary the tree counts, tree depths, and feature counts. We then took the generated models and converted them into decision tree automata to run on the Automata Processor. For the MNIST data, we chose to represent the handwritten digit input data with a 784 byte wide feature vector, one per pixel. For the Twitter data, we used TF-IDF (Term Frequency, Inverse Document Frequency) vectorization with an experimentally-determined 1600 feature size.

3.4.3 CPU Evaluation

We determined the CPU throughput values of our RF models by using Scikit-Learn's Random Forest implementation. While benchmarking CPU performance using multi-cores, we found that the performance varies depending on the hardware configuration and the algorithm's parallel efficiency. We chose to evaluate against a single thread of the Intel Xeon CPU E5-2630 v3 @2.4GHz processor.

3.4.4 Automata Evaluation

Our generated Random Forest models were converted into our automata representations and loaded onto the AP with Micron's SDK. Knowing the number of STEs per automata, the feature vector size, and the number of trees in the ensemble, we could calculate the throughput of our models on functioning AP hardware. There are 16 rows of STEs per block, 192 blocks per AP chip, 8 chips per AP rank, and 4 ranks per AP board. The input symbol rate was expected to sustain 133 MegaSymbols -per

second. If the Random Forest model fits on a single rank, we use inter-rank multi-streaming to increase our throughput to 4x a single rank throughput. If the model is small enough to fit into two chips in a rank, we use rank core multi-streaming to achieve an additional 4x speedup, with an effective throughput of 2.128 gigasymbols per second!

3.4.5 Results and Discussion

Random Forest Model Parameters and Accuracy

The experimental results on Twitter data show that the classification accuracy increases with the leaf count. We found that the maximum accuracy for our model saturated at 72% with 800 leaves per tree as shown in Table 3.1. We also found the classification accuracy to increase from 5 to 40 trees, but no more significant increase of accuracy beyond that count.

Table 3.1: Automata Size vs Accuracy and Throughput for Twitter Results

Trees	Leaves	Accuracy	AP Throughput (k Pred/Sec)	CPU Throughput (k Pred/Sec)	AP Speedup
5	40	66.9%	14400	154	93
10	40	67.5%	8130	129	63
20	40	67.7%	5360	93.4	57
40	40	68.0%	3750	58.5	64
5	600	70.4%	2010	118	17
10	600	71.4%	1530	86.4	18
20	700	71.7%	385	51.5	7
40	700	71.9%	194	32.4	6

We performed the same exploration for the MNIST dataset models. Our results show that increasing the number of leaves per tree in the ensemble has a similar effect as with Twitter data, and we found our elbow was around 10000 leaf nodes per tree. Unlike with the Twitter dataset, MNIST data models had a significant increase in accuracy when increasing the number of trees per model from 5 to 160. Our highest experimental accuracy was calculated to be 97.1% and was computed with 160 trees with 4500 leaves per tree as shown in Table 3.2.

Table 3.2: Key data points of MNIST Results

Trees	Leaves	Accuracy	AP Throughput (k Pred/Sec)	CPU Throughput (k Pred/Sec)	AP Speedup
5	50	82.2%	13200	337	39
10	50	86.1%	5980	242	25
20	50	87.8%	4170	150	28
40	50	88.7%	3350	86.5	39
80	50	89.2%	2940	46.4	63
160	50	89.6%	1350	25.0	54
10	500	93.3%	2480	205	12
20	500	94.3%	1160	125	9
40	750	95.2%	420	68.0	6
80	1250	96.0%	111	34.3	3
20	4000	96.1%	129	98.9	1.3
40	4750	96.7%	55.0	51.5	1.1
80	5000	96.9%	25.0	26.6	0.9
160	5000	97.1%	12.2	13.5	0.9

Throughput vs. Accuracy

Generally, there are design tradeoffs that can be made between throughput and model accuracy for ML inference. Random Forest models with fewer and shallower trees can achieve higher throughput, but below a threshold, they achieve lower accuracy. Adding additional trees and training them to be deeper increases the accuracy to the model's saturation; adding any additional resources beyond this point just reduces the efficiency of the model and can lead to over-fitting. The goal of model optimization is to find the trade-off between these parameters that maximizes throughput, while still achieving the required level of accuracy.

Figures 3-10 and 3-11 show that throughput is significantly affected by the number of trees in the Random Forest ensemble. As we discussed above, we saturate our accuracy with 40 decision trees, and therefore adding any more is unnecessary. With the same number of trees per model, the AP consistently performs with higher throughput than a single-threaded CPU on Twitter data. Figure 3-11 shows that, on MNIST data, the AP outperforms the CPU in most of the cases. With the number of trees greater than 20, and a large leaf number per tree (over 4000 leaves per tree),

the throughput matches.

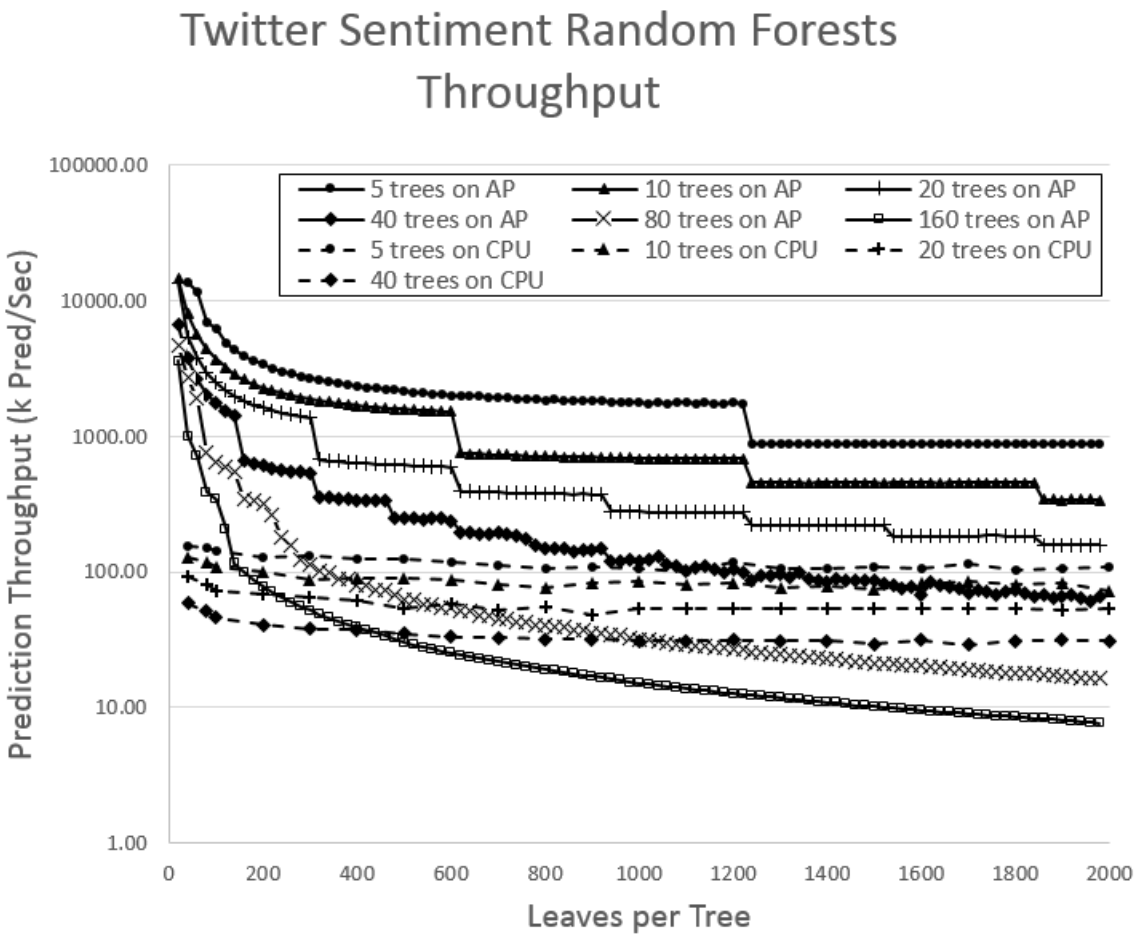


Figure 3-10: Throughput of Twitter Random Forest as a function of number of trees an leaves.[92]

The AP architecture allowed us to multi-stream by 16x if the full model fits into two of the 32 AP chips. As the model size increases beyond this threshold, this multi-streaming factor is reduced by factors of two (Table 3.1, Table 3.2) . The steps in the graph indicate the model dimensions where the hardware cannot sustain the multi-streaming factor. Future generations of the hardware will be able to fit larger models, therefore flattening the throughput curve.

For the Twitter models, the Random Forest implementations on the AP achieve from 2 times to 93 times the prediction throughput of a single CPU. For MNIST, the AP can achieve up to 63 times speed up over the CPU. The speed ups achievable using the AP are more significant with models that have fewer leaves per trees and

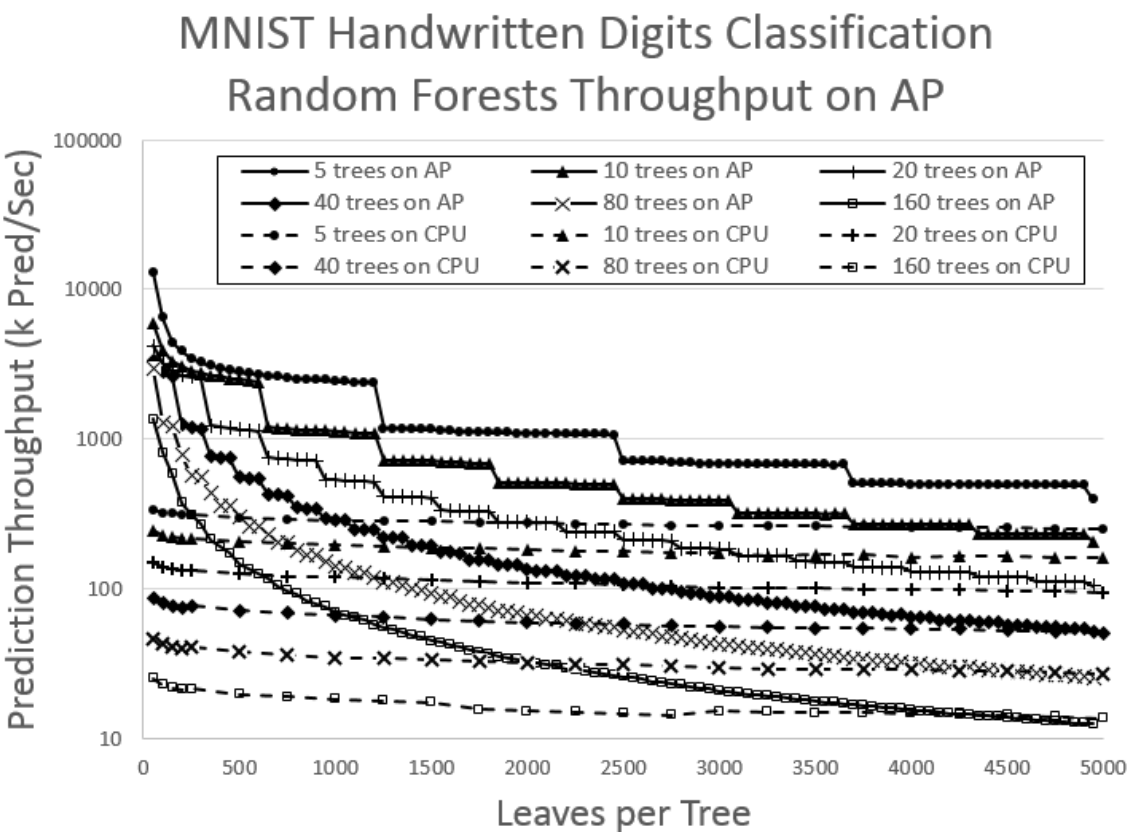


Figure 3-11: Throughput of MNIST Random Forest as a function of number of trees an leaves.[92]

fewer trees per forest.

The AP is a massively parallel device. With smaller Random Forests models, especially for models with lower numbers of leaves per tree, the AP's advantage of massive parallelism can process hundreds of trees simultaneously. For the smaller models, we were able to achieve results with up to 93 times speedup against a single CPU thread. The AP's advantage decreases as the number of leaves per tree increase. With significantly decreased parallelism, a higher frequency CPU can reach similar performance. With these properties in mind, it is important to focus on compacting ensemble models on the AP to maximize performance.

Chapter 4

Automata Optimizations

In chapter 3, we showed that decision tree ensembles could be transformed into a set of Finite State Automata (FSA) by breaking the trees into all paths and representing each path with a separate FSA. By mapping these FSAs to spatial architectures, we showed that we could achieve significant parallelism that von Neumann architectures could not provide. One limitation of our approach is that we map thresholds directly to states, limiting the number of unique thresholds per feature to 255, or the Alphabet size of an STE. In this chapter we present a solution to the limited alphabet size by representing super-alphabets composed of multiple states.

As part of our analysis, we showed that input vector size, or the number of bytes that are processed by the spatial architecture per inference, and the capacity requirements of the automata representation were directly related to the inference throughput. If we could reduce the size of the input stream size by half, we would achieve 2× the throughput. If we could represent our ensemble on the spatial architecture in half the spatial resources, it is possible to get 2× the throughput with multi-streaming as well. This chapter will also explore the space/time tradeoff of several key automata compaction algorithms that we devised that are applicable for decision tree automata as well as other automata applications.

4.1 Scaling Alphabet Size

One limitation of the decision tree automata (DTA) we presented in Chapter 3 is the upper bound set on the number of feature thresholds DTAs can support. Because the STE bit columns on the Automata Processor are 256 bits tall, our approach allows a maximum of 255 unique thresholds per feature. Although this is sufficient for smaller models or models with less threshold variation like Random Forests, models with higher threshold variation including boosted regression trees quickly pass this limit. To demonstrate this, we trained a Random Forest on MNIST with 20 trees, 800 leaves each, and 200 features. This model when transformed into our DTA representation contains $20 \times 800 = 16000$ automata. Figure 4-1 shows the distribution of unique threshold counts. Because none of the features have more than 254 unique thresholds, this model can be represented with DTA without modification.

We then trained a boosted regression tree with 1000 decision trees, 10 leaves each, and 200 features. This model is smaller than our RF model with $10 \times 1000 = 10000$ automata. Figure 4-2 shows the distribution of unique thresholds. One feature requires more than our allotted space of 255 thresholds. Furthermore, because of the significant difference between thresholds counts among the features, Automata Folding as presented in Chapter 3 would result in poor utilization of memory resources. Optimization Algorithm 3.1 would yield a fold where the STE with the large feature would fill up the STE while the remaining STEs would be underutilized.

4.1.1 Single- Versus Multi-Character Character Sets

Wang et al.[109] use automata to find frequently associated items in large databases. They recognize 16-bit symbols by concatenating two 8-bit STEs and representing 16-bit super-symbols with two 8-bit symbols. This approach works in the case where the automata states only accepts one symbol from the super-alphabet, but fails with sets of symbols because of a phenomenon we call *cross product contamination (CPC)*.

CPC occurs when an arbitrary set of items from a super-alphabet is represented by concatenated states. Figure 4-3 shows how Wang et al.[109] represent a 16-bit super-

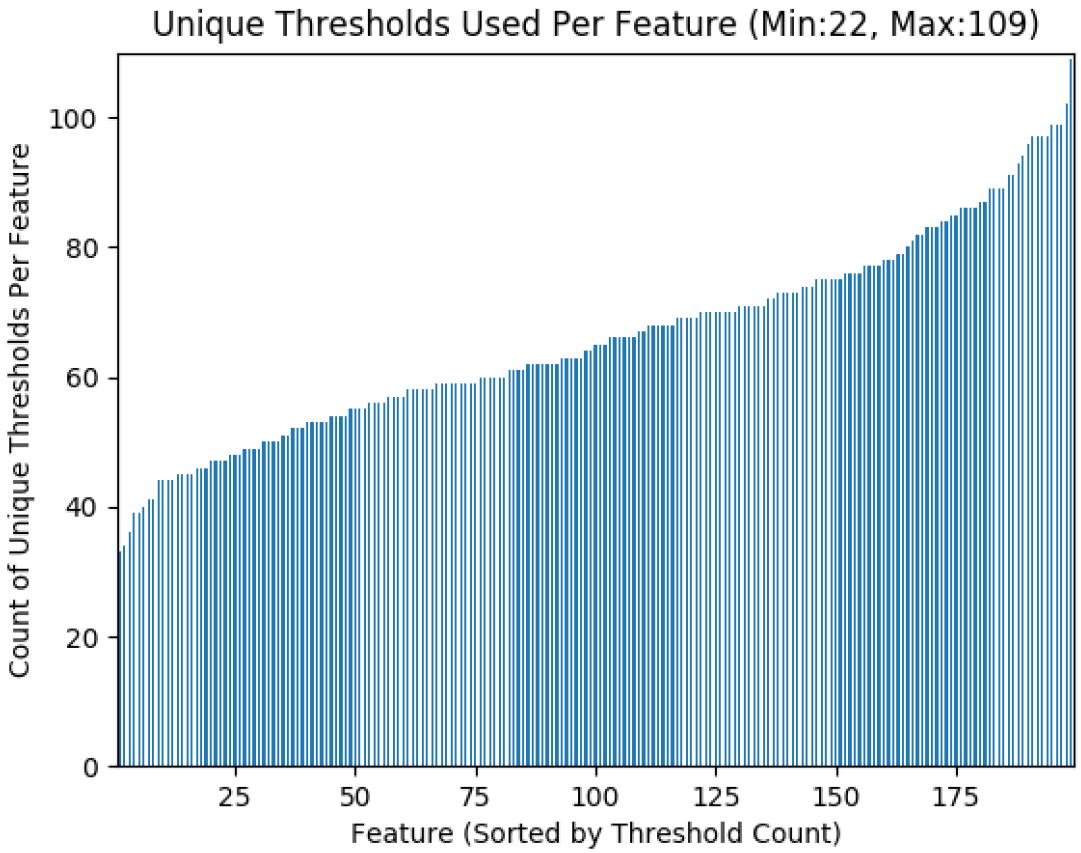


Figure 4-1: Number of unique thresholds per feature for a Random Forest with 20 decision trees, 800 leaves per tree, and 200 features.[106]

symbol by concatenating two state elements. To represent an item in this larger alphabet, two symbols need to be processed; the first for the most-significant-byte (MSB), and the second for the least-significant-byte (LSB). This technique works if a single item is to be accepted, as shown by the first row under the STEs. In this case, the two STEs are accepting the symbol 0000000100000001, or 257 in deciminal. The input would then be streamed in as 00000001 followed by 00000001.

The second row shows a situation where a set of two super-symbols from the 16-bit alphabet are meant to be accepted by the two concatenated states: 0000000100000001 and 0000001000000010. Both of these super-symbols are broken into LSB and MSB and represented in their respective STEs.

This technique fails, because the two concatenated STEs also accept all cross-products between the MSBs and LSBs character sets. In this case, 0000000100000010

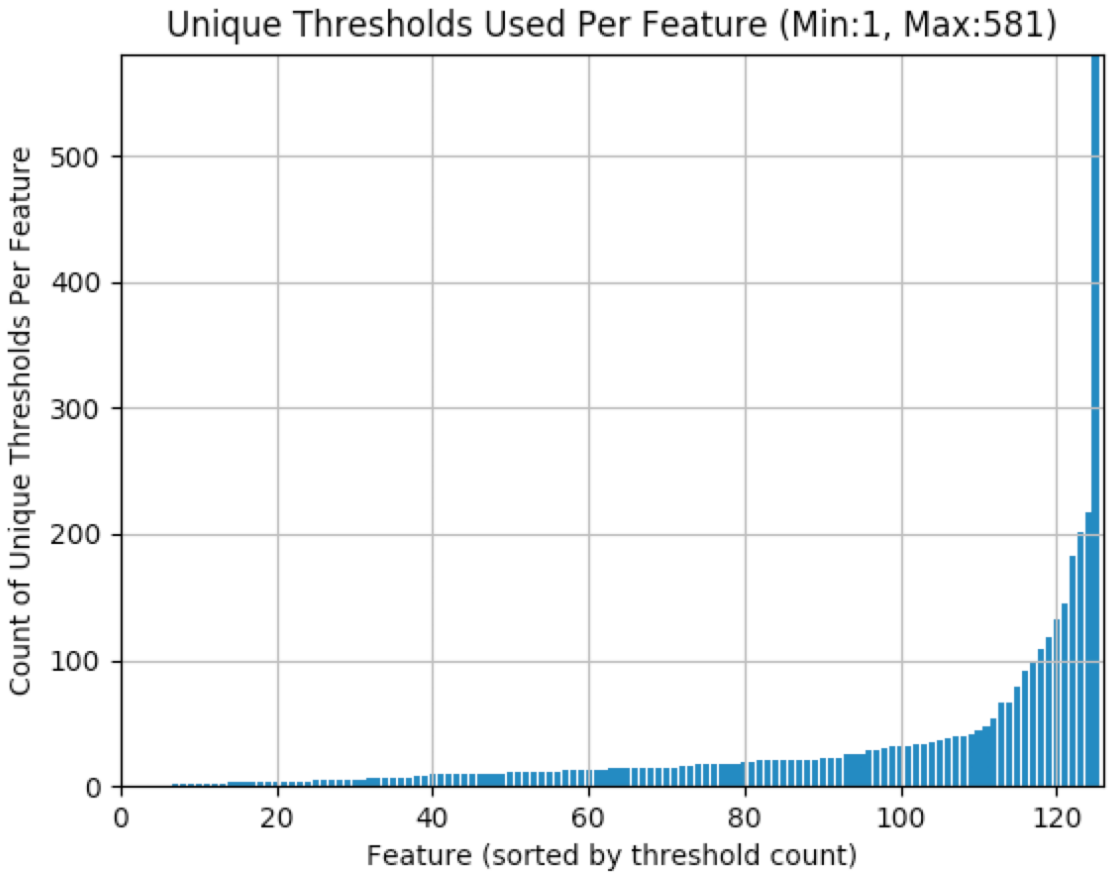


Figure 4-2: Number of unique thresholds per feature for a boosted regression tree model with 1000 decision trees, 10 leaves per tree, and 200 features.

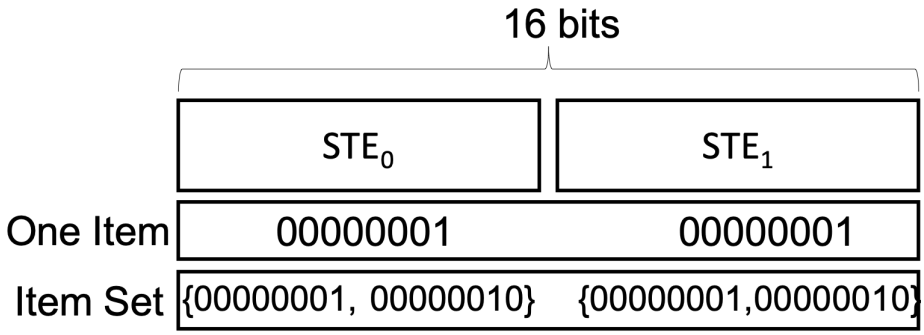


Figure 4-3: Representing 16-bit super-symbols for one item, and cross contamination with representing item sets.

and 0000001000000001 would also be accepted, even if they weren't intended to be part of the accepted inputs. We present a solution to deal with larger alphabets by using a *One-Hot Automata Encoding*, and also introduce a special case for contiguous

ranges in our *Two-Hot Automata Encoding*.

4.1.2 One-Hot Encoded Super-States

One-Hot automata encoding is based on the one-hot binary encoding representation. It works by effectively combining the bit columns of STEs into one large bit column. Unlike the work of Wang et al.[109] and their bit-level encoding where the address space grows exponentially with STE count and input symbol count, our approach grows linearly in both. In exchange, our One-Hot encoding can represent any subset of the entire address space without CPC.

In order to avoid CPC, One-Hot encoding enforces a character encoding with an empty valid cross product. It requires that all input super-symbols be represented with only one value symbol, and that all other symbols be represented with *DONT CARE* symbols. To demonstrate One-Hot Encoding, we show a simple automaton that accepts symbols in the range $701 - 762$ and the input symbol stream that corresponds to 701.

Figure 4-4 shows an example of a super-state STE representation that uses one-hot encoding to represent all values between 701 and 762. It requires three STEs. The input super-symbol 701 is then represented with three symbols. Index 701 maps to the third STE, so the first two are set to the *DONT CARE* symbol 255, and the third is set to the value that maps to 701, 193.

This encoding technique is applicable to any automata application with large alphabets and can be dropped into existing automata designs by replacing single automata states with these chains of states. The number of STEs as well as the runtime of inference is linear in the size of the feature address spaces.

4.1.3 Two-Hot Encoded Super-States

Our second encoding technique requires that all items accepted in the character set be in one continuous range. This assumption does not generalize across all automata, but it is applicable to decision tree automata, because each automaton represents one

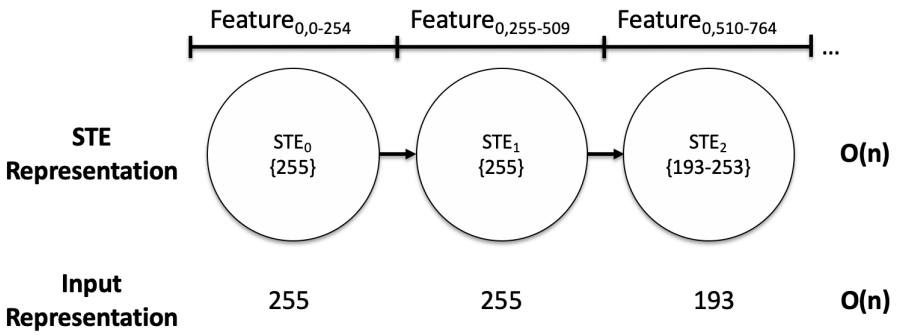


Figure 4-4: One-Hot Encoding.

continuous partition in feature space.

Two-Hot encoding works by representing an address space in a two-dimensional grid where one dimension is represented by one STE, and the other by a second. Figure 4-5 shows an example of a two-hot encoded super-state representation accepting values between 100 and 725 for a feature with 764 unique threshold ranges. To reduce the address space used by each of the two STEs, we find the square root of 764 and represent all thresholds in a square of size 28×28, where one STE represents the X-axis and the other the Y-axis. We index into the square address space shown in blue in column-major order, indexing from the bottom-left up to the top of the square, and then up from the bottom of the next column.

The grid next to the blue block indicates the range that this automaton will accept. The green column represents the first partial column of values that are accepted and the orange column represents the last partial column. The red rectangle between the two partial columns is the set of full columns in the acceptable range.

The three pairs of states shown under “STE Representation" shows how we accept the different parts of the continuous range of values. The first pair of states, shown in green, accept feature values that map to the first partial column and the one in orange accepts those that map to the last partial column. The first state of each colored pairs indexes into the X-axis of the plot and represents the column index. The second state represents the rows in the partial columns that are accepted by the range. The middle two red states accept the continuous rectangle between the first

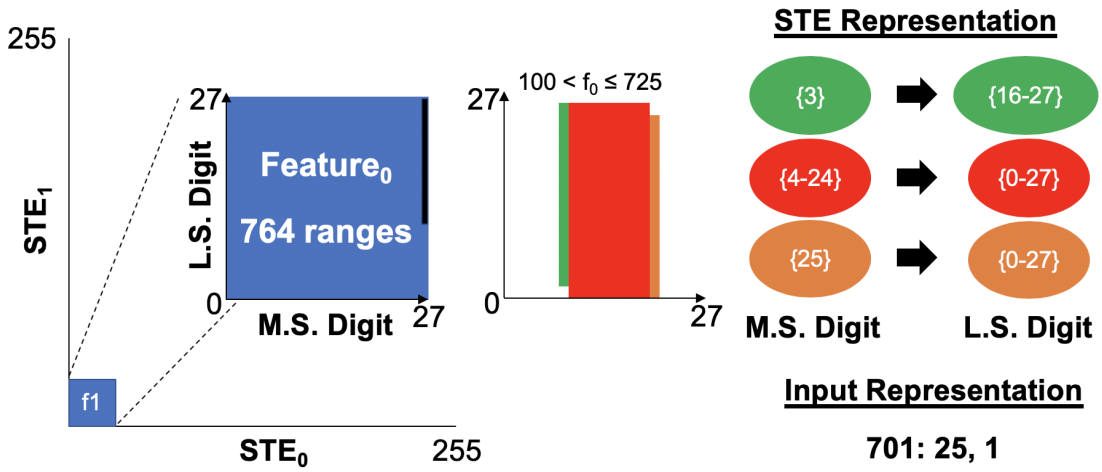


Figure 4-5: Two-Hot Encoding.

and the last partial columns.

This encoding technique requires six STEs to represent the two-dimensional address space, but can represent up to 256^2 unique thresholds; One-Hot encoding would require 256. The input representation, as shown in the figure, indexes into the two-dimensional space. 701 for example would index into the 25th column and the 1st row. The 25th column corresponds to the last partial column and would activate the first orange state. The second symbol, 1, would then be accepted in the partial column range indicating a match.

4.2 One-Hot and Two-Hot Automata Folding

One-Hot and Two-Hot automata encoding allow us to represent arbitrarily large automata alphabets, but our STE utilization is limited if we cannot combine multiple features into the STEs. To accomplish this, we present One-Hot and Two-Hot Automata Folding (AF), extending our original AF approach to these new encoding techniques.

4.2.1 The Grid and Block Abstraction

In order to generalize our Automata Folding approaches, we represent automata using a *Grid and Block Abstraction*. This approach allows us to provide one common interface to discuss encoding techniques. We use this abstraction when describing both One-Hot and Two-Hot Encoded AF.

A *Grid* represents one continuous symbol address space. In the case of One-Hot representation, a Grid represents the address space of one large character set that can be compoosed of multiple STEs. For the Two-Hot representation, a Grid represents the full 256×256 address space provided by the two STEs. We showed that with Automata Folding we could use an address space to represent multiple, smaller address spaces. To address those independent and non-overlapping address spaces, we use the Block abstraction. For the One-Hot encoding, we can have multiple blocks assigned to 1 Grid, as long as the combined address space does not exceed the size allocated to the chain of STEs. For the Two-Hot encoding, we can fill the two-dimensional Grid with feature Blocks.

4.2.2 Feature Permutations

Our original Automata Folding algorithm presented in Chapter 3 assumes an ordering of features, simplifying the optimization algorithm to one where only the number of states serves as the independent variable. Although all automata need to process each feature simultaneously in the same order, this global permutation of feature access is entirely arbitrary.

The flexibility to move these features around into a different permutation opens the door for improved spatial efficiency, by allowing us to combine features and states in such a way to minimize the number of total states required to represent our automata, and therefore total number of spatial resources required. This optimization comes at a cost. By introducing another dimension, that of ordering the features, we increase the complexity of this problem to an NP-Complete bin-packing problem. In order to compute more efficient packings of features in a reasonable amount of time, we utilize

a polynomial-time bin packing heuristic.

4.2.3 One-Hot Automata Folding

One-Hot AF works in a similar way to our earlier AF implementation presented in Chapter 3, but with support for *large features* (those with > 255 threshold ranges) and by reordering features, the algorithm can achieve significant improvement in memory utilization.

To accommodate features with more than 255 unique threshold values, we select all large features and split their address spaces into sub address spaces (Blocks) using our One-Hot encoding. These large features are then assigned to their own states, and are not part of the circular access pattern of Automata Folding. We assign these large features to the first few states in the automata, and represent the first few feature values in the input feature stream to these large features.

The features with less than 255 unique threshold values are combined into fewer automata states with our One-Hot Automata Folding algorithm. The goal is to make a loop of as few states as possible. In our improved algorithm, we use a variation on the best fit decreasing bin packing heuristic; in this case, an automata state is treated like a bin of features, where the size of the feature is the number of thresholds associated with that feature + 1. We sort the features by number of unique threshold values, from the feature with the most thresholds, to the feature with the fewest. We use a min heap to fill the most empty bin in the set of bins with the next largest feature, and add features until either we are out of features to add to the bins, or the bins have overfilled. If the bins are overfilled; we add another bin and repeat. Algorithm 1 shows how our bin packing algorithm works.

After computing our One-Hot Folding algorithm, it is possible that the bins have differing numbers of features associated with them. This is a problem, because Automata Folding requires a round-robin access pattern of the states, where the automata are effectively loops of states accessed one after another. To balance out the number of features per state, we introduce a balancing algorithm. Our balancing algorithm first determines the bin with the fewest number of features, and pops off

```
Result: Heap of lists of features such that their combined address space is less
        than capacity
num_stes = 1;
sort(features, key=len, reverse=True);
while True do
    # Fill heap with n empty bins for i in range(num_stes) do
        heap.push(([], 0));
    end
    # Fill bins first-fit decreasing
    for feature in features do
        bin, old_size = heap.pop();
        new_size = old_size + len(features.thresholds);
        bin.append(feature.thresholds);
        heap.push((bin, new_size));
    end
    # Make sure we haven't over-filled
    if max(heap) ≤ 255 then
        break;
    end
end
```

Algorithm 1: Using Bin Packing to efficiently pack feature address spaces in as few states as possible

features from all other bins to enforce a consistent number of features per bin. It then proceeds to add the next biggest extra feature to the least empty of the remaining bins, adding that bin to a timeout list, enforcing the requirement that all bins have a feature count within 1 of all of the rest.

We present a proof[93] that packing bins such that the number of items within each bin is within 1 of all of the rest is an NP-Complete problem, and therefore there is no known tractable perfect solution. This means that we need an approximate solution.

Bin Packing The bin-packing problem is as follows: Given a list of objects of various volumes $(O_1, \ldots O_n)$, and a bin capacity V, assign objects to bins in such a way to minimize the number of bins required to contain all items in the list. This problem is famously NP-Hard.

k-balanced Bin Packing Call $|B_i|$ the number of items in bin i. A packing solution is k-balanced if the difference in the item count of the bin with the maximum number of items and that with the minimum number is $\leq k$; in other words, if $MAX_i|B_i| - MIN_i|B_i| \leq k$. Thus the k-Balanced bin packing problem is as follows: Given a list of objects of various volumes$(O_0, \ldots O_n)$, a bin capacity V, and an item-balance threshold k, assign objects to bins in such a way to use the minimum number of bins such that each is k-balanced.

Theorem 1. 1-*balanced bin packing is NP-Hard*

Proof. We will show how to use a polynomial time solver for 1-balanced bin packing, BP_1 to construct a polynomial time solver for the traditional bin packing problem BP, showing a reduction to BP.

Consider if there is an object in an optimal bin-packing solution which has zero (or sufficiently small) volume. A solution constructed by removing this small object from the original solution will also be optimal. In the case that the latter solution used fewer bins, we could construct a more optimal solution to the original bin packing instance by taking this new smaller solution, then adding the small object to any arbitrary bin.

Using this idea, if we had a 1-balanced bin packing instance that was "padded" with some objects having zero (or sufficiently small) volume, then an optimal solution could be converted into an optimal standard bin packing solution by removing these small "padding" objects. In this way we can reduce a polynomial time solver for bin packing, BP, to a polynomial time solver for 1-balanced bin packing BP_1 by taking the input to BP, that is a list of n objects, represented by their volumes, and padding this solution with m objects of zero (or small) volume. In this case, if BP_1 runs in time $O(n^p)$ for some p, then this algorithm for BP will run in time $O((n+m)^p)$. Now all that remains is to define a sufficiently large m (it is impossible for m to be too large). As long as m is polynomial in n then this solution to BP is also polynomial.

Any unbalanced bin-packing solution will have at most n bins, one for each item. Also, each bin in an unbalanced bin-packing solution may contain at most n items

(all in one bin). Therefore, naively, the maximum unbalance in a bin-packing will be n across $n-1$ bins. This means that there always exists a choice of $m < n^2$ to pad an unbalanced packing to be 1-balanced, so $m = n^2$ is sufficient. For this choice of m, the time to solve bin packing is $O((n+n^2)^p) = O(n^{2p})$, so bin packing can be solved with quadratic overhead over 1-balanced bin packing. □

4.2.4 Two-Hot Automata Folding

We extend our One-Hot Automata Folding (AF) technique to Two-Hot encoding. Figure 4-6 shows how we can combine multiple Two-Hot encoded features into the six STEs by aligning the feature Blocks on a diagonal. Like with the One-Hot approach where the feature Blocks occupied non-overlapping parts of the full STE address space, Two-Hot AF requires that the blocks not overlap with both dimensions, resulting in a diagonal line of blocks in the Grid.

One-Hot AF scales spatial resources and runtime linearly with the feature range count. Our Two-Hot AF technique scales spatial resources and runtime by the square root of the feature range count. Our approach does this by representing feature threshold address spaces with Two-Hot automata encoding as square *Blocks* in a two-dimensional *Grid*.

We use the same first-fit decreasing algorithm as with One-Hot Automata Folding, but this time each feature's address space size is the square root of the number of thresholds, as we've transformed the array of thresholds into a square with side lengths of length square-root. This results in a total number of states:

$$NumStates = 6 * \frac{1}{255} \sum_{f=0}^{N} \sqrt{Thresholds_f}$$

Figure 4-7 shows the resulting automata on the right. The black arrows show how the states are accessed in a round-robin fashion until the delimiter character, 255, is seen. If the automata have recognized all pairs of feature symbols up until 255, the automaton will accept the input as the tree traversal path it was configured for. There is no such thing as free lunch, so our approach has several disadvantages. For

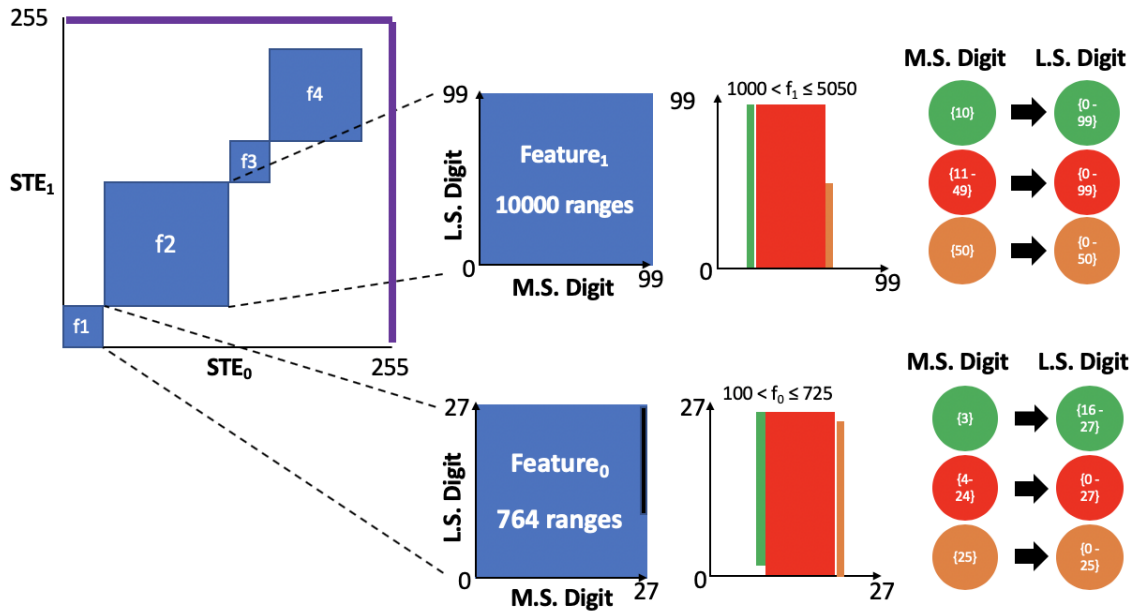


Figure 4-6: Encoding features with 2d encoding.

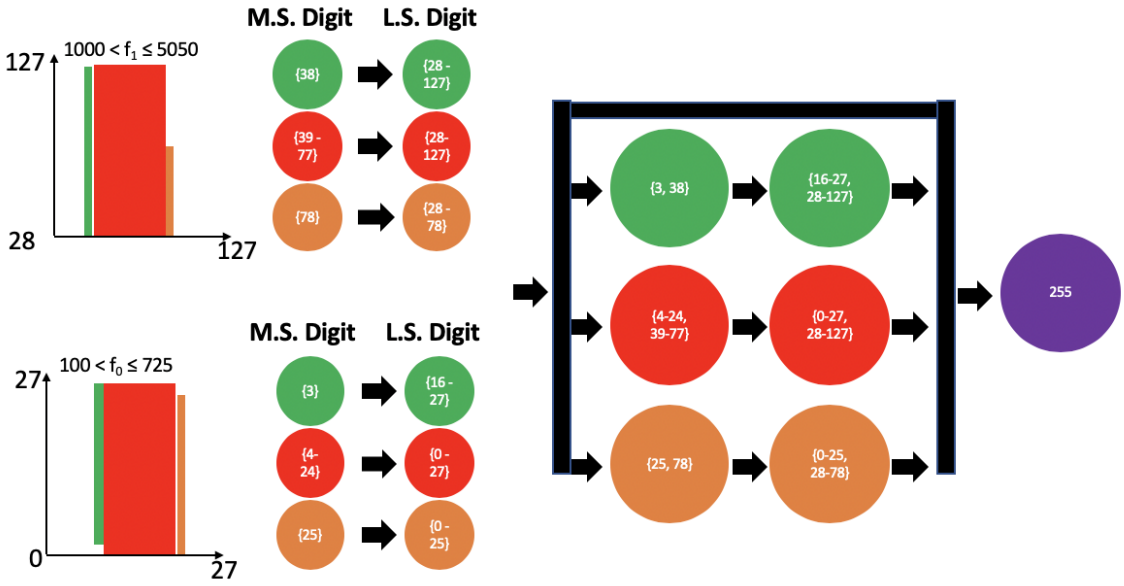


Figure 4-7: 2D-encoded Automata Folding.

one, six states are now required per Grid. If a full Grid is not used, this is a significant overhead. Also, each input feature is now represented with two symbols. In the case of 1D encoding, if the feature has fewer than 255 unique thresholds, one symbol is sufficient to represent the feature; for 2D encoding, a minimum of 2 symbols are used per feature:

$$NumSymbols = 2 * \sum_{f=0}^{N} \left\lceil \frac{\sqrt{Thresholds_f}}{256} \right\rceil$$

The resulting three pairs of states represent any continuous range of values in the square sub-space of the 255×255 full address range. The overhead for a single feature is significant, at $6\times$ the STE cost, but with automata folding and the ability to loop back on the states, we can achieve a much more efficient spatial representation of the automata.

Let us consider the extremes when using Two-Hot Automata Folding and find the feature cardinality that makes the spatial overhead worthwhile. For our best-case scenario, let us consider a feature that contains 255^2 threshold ranges. In this case, we can represent 65025 feature ranges in 6 states, whereas the original automata folding algorithm would have mapped this feature as a Large Feature to 255 states. This gives us an increased spatial utilization of $255 \div 6$ or $42.5\times$!

The other extreme is representing 127 features all of cardinality 2. This would correspond to 127 unique features that each contain a single split node. The resulting effective address space size is 127 * 2 = 254 per 6 states, or 42.3 ranges per STE, whereas our One-Hot folding technique would be able to fit all of these features into a single STE, achieving a effective address space size of 254 per 1 state, or 254 ranges per STE, a $6\times$ increase in utilization! It therefore becomes necessary to find the point of inflection, where feature address spaces are large enough to warrant the space cost.

This approach can be generalized to an arbitrary dimensionality, further increasing our spatial efficiency, but with each additional dimension, the number of STEs to represent boundaries increases exponentially as well, effectively introducing higher overhead per N-dimensional grid. We leave this generalization to future work.

4.2.5 Evaluating One-Hot versus Two-Hot Automata Folding

AutomataZoo[106] is an automata processing benchmark suite composed of 24 different benchmarks that span several domains. One of those benchmarks contains three different Random Forest models that classify 28x28 pixel images into the nu-

merals 0-9. These models were trained on the MNIST[46] digit recognition database.

To compare the One- and Two-Hot Automata Folding techniques, we evaluate the performance of both AF techniques on all three Random Forest model variants, as shown in Table 4.1. We then determine the number of states that the resulting automata contain, as well as the number of input symbols required per inference.

Variant	Features	Max Leaves	States	Accuracy	Runtime
A	270	400	248k	93.37	1.35x
B	200	400	248k	92.91	1.0x
C	200	800	992k	93.85	1.0x

Table 4.1: Random Forest benchmark variant trade-offs from the AutomataZoo Benchmark Suite[106].

Table 4.2 contains the results of using the One-Hot and Two-Hot AF on all three variants. Although we have shown that our Two-Hot AF approach requires assymptotically fewer states and input symbols as the number of thresholds per feature increases to infinity, for both variants A and B, One-Hot AF results in fewer number of states used and smaller input streams. Only in the larger variant C did we achieve a space reduction. In this largest and most accurate model we were able to encode the automaton with 29% fewer state elements using Two-Hot AF.

Variant	One-Hot States	Two-Hot States	One-Hot Input	Two-Hot Input
A	248k	304k	271	541
B	248k	256k	201	401
C	992k	704k	201	401

Table 4.2: Number of automata states per Random Forest model.

These results indicate that Two-Hot AF is preferential with very large models because of the large constant overhead imposed on the input length and automata size. We showed that Two-Hot AF requires six state elements per super-state, and that this overhead is significant, making Two-Hot AF inefficient for smaller models. To encode the input stream, we also require an additional symbol per feature, one for each dimension. This halves our throughput for our C Variant model.

4.3 Further Optimizations

4.3.1 Compacting The Input

Automata computing works by processing ordered input stream symbols sequentially; state transitions are made on these symbol values as a function of the previous active state and the input symbol. If the input stream size can be reduced, a linear throughput increase is achieved. This motivates approaches like *automata striding*[62], where multiple logical input symbols are combined and represented by a single input symbol. Striding works by representing multiple symbols from a small alphabet with one from a larger alphabet. As an example, the bioinformatics alphabet $\Sigma_0 = A, C, G, T$ can be represented with a 2-bit alphabet. Some existing automata approaches[77, 89] represent each base with an entire 8-bit symbol. This is inefficient because six extra bits are never utilized by the automata computation.

Wadden, et al.[106] and Becchi, et al.[7] use bit-level automata striding, representing multiple sequential base inputs within one symbol. With our example alphabet from above, we could use 4-striding, where one input symbol can represent 4 2-bit bases. This then allows us to represent 4 sequential bioinformatics inputs with a single 8-bit symbol by breaking the 8-bit space into 4 non-overlapping 2-bit symbols. This is illustrated in Figure 4-8 where the four nucleotides shown in the bottom row map to four separate 2-bit slots in one 8-bit symbol. This allows us to stream four logical symbols per symbol, achieving a 4× increase in throughput.

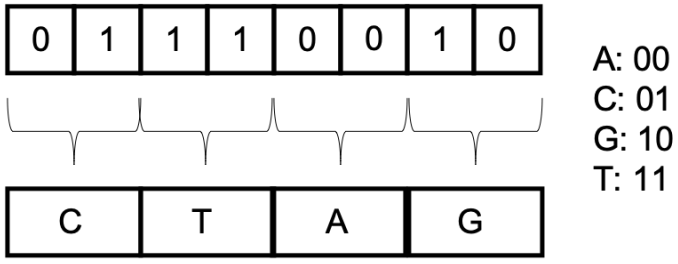


Figure 4-8: Combining four nucleotide characters into one 8-bit symbol with bit-level automata striding.

One limitation of this approach is the tendency to have growing dead zones in the bit-wise address spaces with larger alphabet sizes. A dead zone is a range of STE

values that do not have a symbol mapping. Effectively, they are wasted space. As the size of the alphabet increases, so does the distance between powers of two, and so does the distance between a random alphabet size and it is nearest power of 2. As an example, let us consider the Bioinformatics alphabet $\Sigma_1 = A, C, G, T, X$, where X is an unknown symbol. This special character is commonly added to a sequence of nucleotides if the aligner is not confident of the identity of a base. Using bit-level striding we would require 3 bits to represent the 5 characters. In our 8-bit symbols, we would only be able to represent two of these characters in one 8-bit symbol.

4.3.2 Feature Compression

Our approach is different from bit-level automata multi-striding in that we can represent a smaller representation than bit-level striding. Our flexible automata encoding gives us the flexibility to represent these encoding techniques with less overhead.

To demonstrate our approach, let us consider compressing Σ_1 by representing as many characters as possible from the alphabet in one 8-bit symbol. There are five symbols in Σ_1. We find the floor of the fifth root of 256, 3. Unlike bit-level striding which can only represent two Σ_1 characters per symbol, we can fit 3. We do this by representing the full cross of three symbols in a cube space as shown in Figure 4-9 and assigning each location in the address space a unique label. These labels can then be used in the input stream and STE character sets in the same way as one would use bit-level striding symbols.

4.3.3 Logarithmic Automata Search

If all automata do not fit on the available spatial hardware, one approach is to buy more hardware! Another is to run automata in sequential batches on the spatial architecture. Although Hybrid Automata Folding increases the number of these automata that can be represented on the fabric, the number of automata is still exponential in the depth of the decision trees, resulting in a $O(2^d)$ runtime. We bring this runtime down to $O(d)$ by partially reverting recursive partitioning.

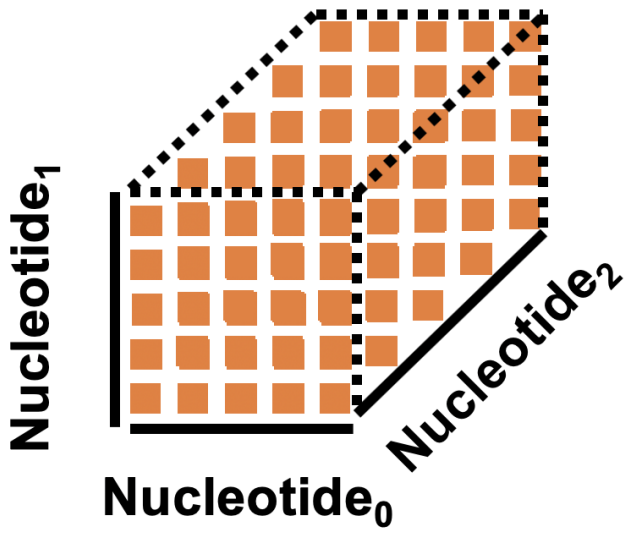


Figure 4-9: Combining three nucleotides from $\Sigma_1 = A, C, G, T, X$ into one 8-bit symbol by representing them as unique addresses in a cube.

Figure 4-10 shows how automata are combined, by agglomerating decision tree branches, forming new branches that represent the union of their constituent branches. These *fuzzy automata* are then loaded onto the spatial architecture and evaluated against the input feature vector. Only the *fuzzy automata* that contain the accepting branch will report. The fuzzy automata that make up this set are loaded and this process is repeated until the a leaf node is reached.

We represent these unified branches with *fuzzy automata* that are then loaded onto the hardware. The input is streamed in, and only the fuzzy automata that contain the valid chain automata report; those constituent chains are then loaded, and this process is continued all the way down the chain, for all trees of the ensemble. This algorithm brings the runtime from linear in the automata count, to the logarithm of the automata count.

One limitation of this approach is the significant overhead required to reconfigure the spatial architecture. To reduce the impact of this overhead, we suggest future work to implement a spatial overlay that allows for rapid partial reconfiguration. This overlay would act in a similar way to Micron's Automata processor, exposing state element character set and routing configuration, while not requiring the significant

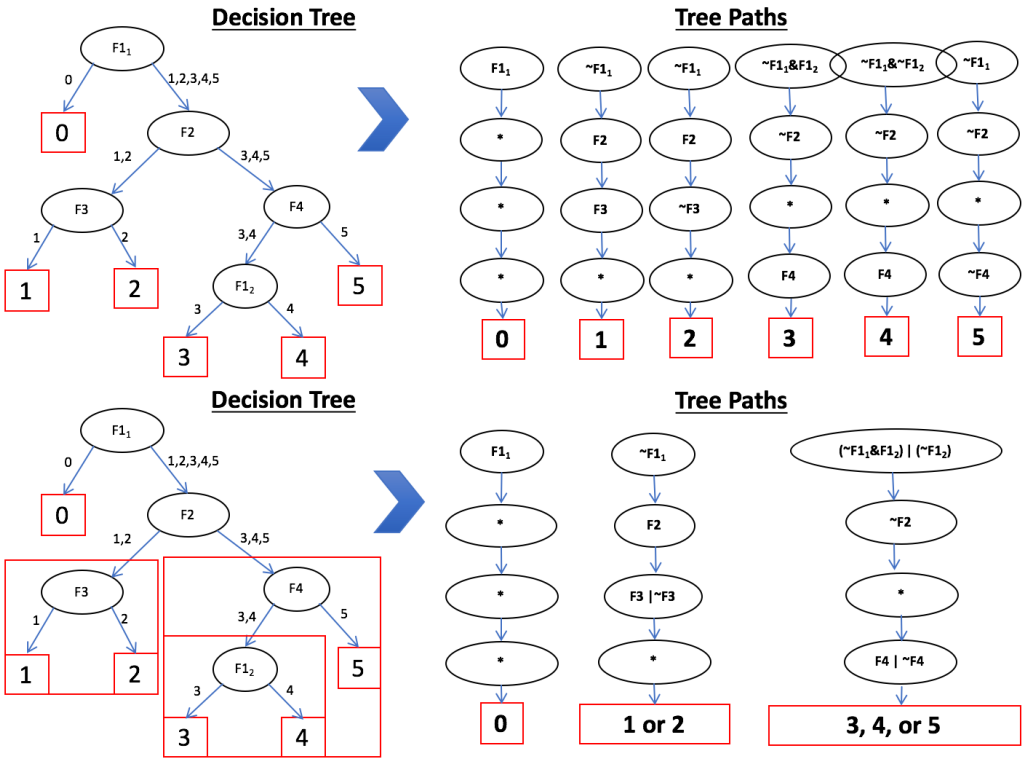


Figure 4-10: Clustering chain automata into fuzzy automata.

overhead of complication and place-and-routing.

Chapter 5

Benchmarks and Tools

One major limitation of many existing automata research projects is the lack of complete and parameterizable automata generation tools. These tools are critical for industry adoption as well as for assisting future research in the automata and architecture domains. We introduce an open source suite of tools called Random Forest Automata (RFAutomata)[37]. This tool suite contains programs to train decision tree ensembles, optimize automata-specific parameters, and generate automata representations of their decision tree-based ML models. We also include tools to evaluate performance and transform data between input types.

In addition to RFAutomata, we also introduce an FPGA synthesis tool called REAPRpp based on previous work by Ted Xie et al. with their REAPR[115] tool. REAPR is an FPGA Automata synthesis tool that converts ANML files into System Verilog files that can be used to target FPGAs. We improve upon REAPR by adding support for debugging[16], adding a reporting architecture[102], and adding additional optimizations like report coalescing. REAPRpp allows researchers and industry developers to deploy decision tree ensemble machine learning models on a variety of FPGA platforms. We target Amazon's AWS F1 instance for performance and accessability, but our generated code is meant to be flexible and should work for any FPGA family.

Finally, our tools and several models generated with these tools were included in two automata benchmark suites: ANMLZoo[104] and AutomataZoo[106]. These

Model Name	Classification/Regression	Framework
Random Forest	Classification	Scikit-Learn
Gradient Boosting Classifier	Classification	Scikit-Learn
Gradient Boosting Regression Trees	Regression	Scikit-Learn
Adaboost Classifier	Classification	Scikit-Learn
Gradient Boosted Regression Trees	Regression	QuickRank

Table 5.1: Supported RFAutomata machine learning models as of version 1.0.

benchmark suites serve as standards to the automata processing community by providing a variety of automata applications to evaluate automata processing architectures. These sorts of benchmarks have become even more important with the end of process node scaling and the industry moving towards application-specific acceleration.

5.1 RFAutomata

RFAutomata[37] is an open-source tool suite for training, optimizing, transforming, and evaluating decision tree automata models. It is open-source and has been released as part of the ANMLZoo and AutomataZoo benchmark suites for generating machine learning benchmarks. RFAutomata supports the machine learning models listed in Table 5.1, and was written to be modular and supports low-effort integration of other decision tree-based ensemble models.

5.1.1 Feature Extraction

RFAutomata includes several ML and architecture tools to assist researchers in design space exploration as well as to catalyze industry adoption of our work. An RFAutomata Feature Extractor is a python program that parses an existing ML dataset and converts it into a Numpy[107] zipped archive file to be used by the training and testing tools in RFAutomata. A Feature Extractor is only necessary if the data is not already in a format that Scikit-Learn can use. In version 1.0 we include two feature extractors: OCRExtractor and MSLRExtractor. OCRExtractor

extracts the pixel features from the OCR datasets[42] into a Numpy matrix as well as extracting the associated classifications into a Numpy vector. These two data structures are then zipped into one Numpy zipped archive file. The MSLRExtractor does the same thing for Microsoft's Learning to Rank datasets[53].

```
python mslrExtractor.py  -i <mslr dataset> -o mslr.npz
```

5.1.2 Training

RFAutomata supports decision tree ensemble models trained using existing frameworks as shown in Table 5.1, but it also includes tools for training models with automata considerations. In order to simplify training decision tree ensemble models, RFAutomata uses a wrapper around Scikit-Learn. This program includes support for hyperparameter-searching as well as dimensional reduction.

We include functionality including providing Scikit-Learn canned datasets including MNIST[46, 24], IRIS[26, 24], Wisconsin Breast Cancer[24], Boston Housing[24], and Diabetes[24] to make training easier as well as generating output files that provide useful information about the trained model's characteristics. To run the training program use the following syntax:

```
python trainEnsemble.py  -c MNIST -m RF -d 10 -n 20 --feature_importance
```

See the available options below for more features that *trainEnsemble* provides.

```
Options:
  -h, --help            show this help message and exit
  -c CANNED, --canned=CANNED
                        A canned dataset included in SKLEARN (mnist)
  -t TRAINFILE, --train=TRAINFILE
                        Training Data File (.npz file)
  -x TESTFILE, --test=TESTFILE
                        Testing Data File (.npz file)
```

```
--metric=METRIC       Provide the training metric to be displayed
-m MODEL, --model=MODEL
                      Choose the model (rf, gbrtc, gbrtr, ada)
--model-out=MODELOUT  Output model file
-d DEPTH, --depth=DEPTH
                      Max depth of the decision tree learners
-l LEAVES, --leaves=LEAVES
                      Max number of leaves of the decision tree learners
-n TREE_N, --tree_n=TREE_N
                      Number of decision trees in the ensemble
-f N_FEATURES, --n_features=N_FEATURES
                      The max number of features used by the ensemble (finds the n
                      best features).
-j NJOBS, --njobs=NJOBS
                      Number jobs to run in parallel for fit/predict
--feature_importance  Dump the feature importance values of the trained
                      dataset as a plot
--explore_parameters  Explore a parameter space for accuracy/runtime
                      tradeoffs
-v, --verbose         Verbose
-r REPORT, --report=REPORT
                      Name of the report file
-p PREDICTIONS, --predictions=PREDICTIONS
                      Predictions made by model to be used to verify
                      the automata implementation
```

Feature Selection

The automata representation of decision trees scales in the leaf count, tree count, and feature count of the models. For this reason, we included functionality for reducing the number of features to be considered when training a decision tree ensemble

with the $-f$ argument. We use a heuristic whereby we train the decision tree ensemble with full features and grab the top-N features in terms of importance. We then transform the feature matrix to include those features, and train the new model. Figure 5-1 shows the computed feature importance values ($--feature\_importance$) for all 28×28 pixels of the MNIST dataset. The intensity of the pixels indicate what the full RF model designates to be the most important features.

We conducted an experiment where we trained a full Random Forest with a varying number of top features on the MNIST dataset and evaluated the accuracy of each model by using 3-Fold Cross Validation. Figure 5-2 shows our results. We found that by setting the number of features to 200, instead of the maximum of 784, we could achieve within 1% of the accuracy, and with 270 features, or 34.4% of the full feature vector, we could achieve within 0.2%. This allows us to significantly reduce both the size of our automata, but also the number of input features we need to stream per inference.

5.1.3 RFAutomata Automata Synthesis

The RFAutomata synthesis tool, called automatize.py, is a Python program that accepts as input a decision tree ensemble model (this can be trained with *trainEnsemble* or imported as a python archive file) and generates an output automata representation in ANML format. The current supported models as shown in Table 5.1 can be pre-trained, but need to be serialized to a pickle file for conversion by RFAutomata.

To run the automata synthesis tool, run:

```
python automatize.py <model file>
```

automatize provides several architecture-specific parameters. Especially for testing small models on von Neumann architectures, we found that unrolling folded automata can yield better performance because it removes the branch loop from the automata. The $--$ *unrolled* argument effectively undoes Automata Folding to generate a series of long chains of set membership operations.

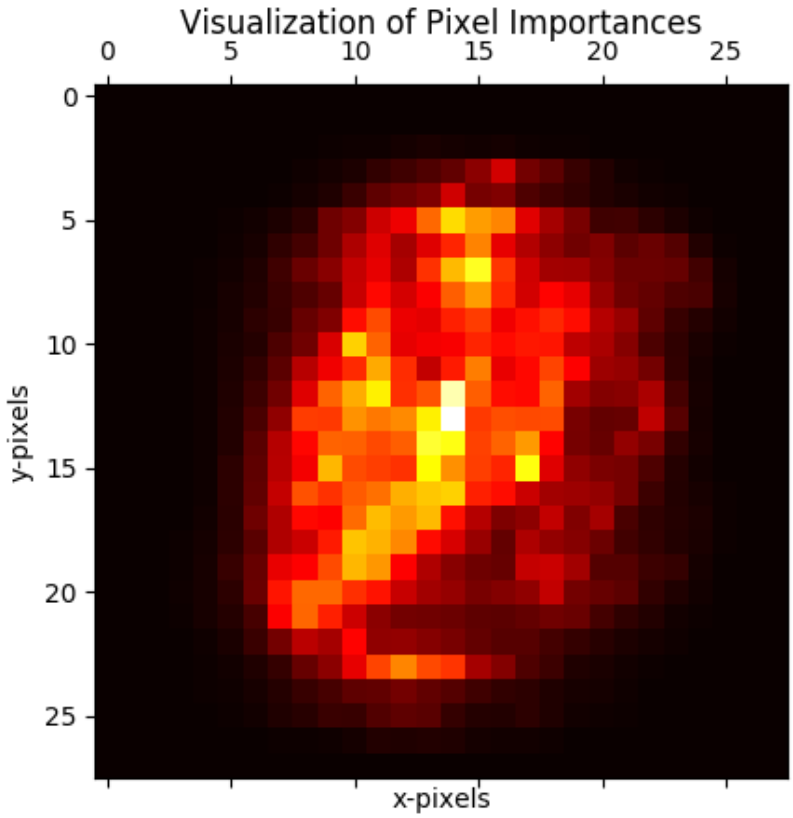


Figure 5-1: Heatmap of the MNIST pixels indicating feature importance in the trained model.

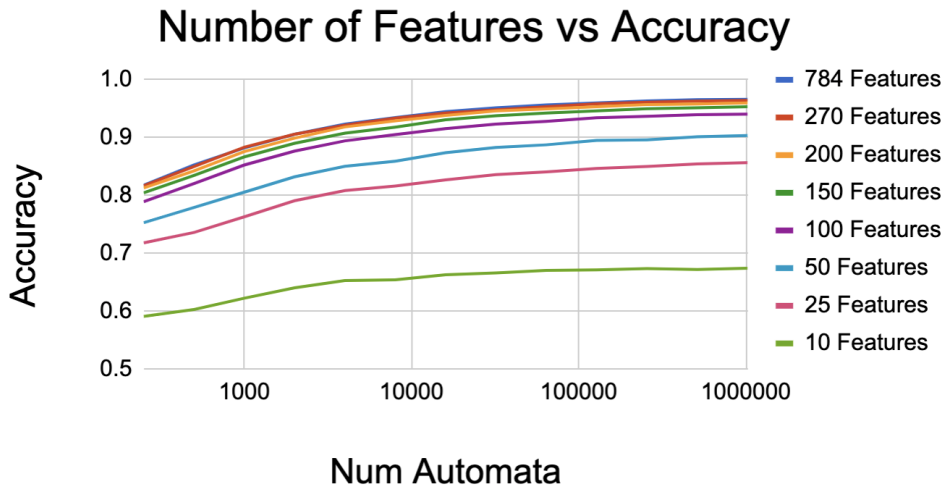


Figure 5-2: Random Forest accuracy on the MNIST training dataset with a varying number of top features.

One important metric to consider when generating decision tree automata is the number of unique thresholds required per feature. This information could be used to decide whether to use One-Hot or Two-Hot Automata Folding. We provide this information with the $--thresholds$ argument which generates a plot as shown in Figure 5-3.

We also support several optimizations. One of them is to merge reporting states at the tree level; another to set the Automata Folding encoding. We know that each tree will only have a single reporting STE per input, and therefore we can combine all leaf nodes with the same output code; in the case of a classifier, nodes that map to the same classification; in the case of a regressor, nodes that map to the same partial sum. This feature is very useful for models like Random Forest where each tree reports one classification per input and all classifications come from a small set of discrete values. By merging all automata from one tree into a small set of reporting states, we can considering reduce the number of output signals that need to be processed. AutomataRF also supports both versions of Automata Folding: One-Hot and Two-Hot with the $--dimensions$ argument.

See the available options below for more features that $rfautomata$ provides:

```
Options:
  -h, --help            show this help message and exit
  -a ANML, --anml=ANML  ANML output filename
  --gpu                 Generate GPU compatible chains and output files
                        **EXPERIMENTAL**
  --circuit             Generate circuit compatible chains and output files
                        **EXPERIMENTAL**
  --unrolled            Set to generate unrolled chains (no loops)
  --short               Make a short version of the input (100 samples)
  -p, --thresholds      Generate a plot of the distribution of threshold counts
  --input_format=INPUT_FORMAT
                        Input file format types:
                        SIMPLE: Space-delimited feature values with newlines
```

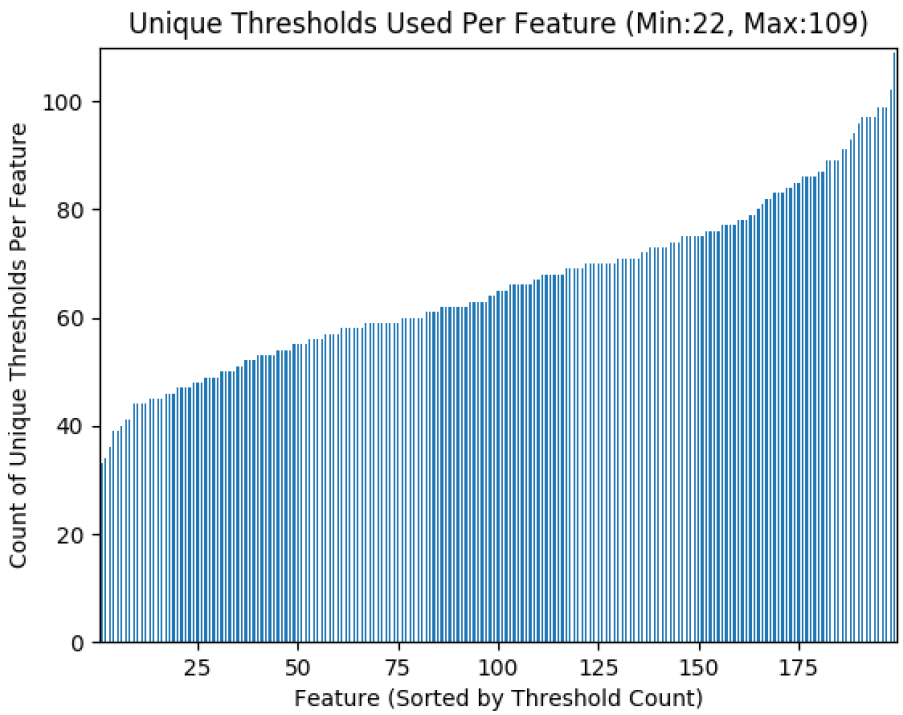


Figure 5-3: Number of unique thresholds per feature for Variant C of AutomataZoo's Random Forest benchmark.[94]

```
                        between vectors (circuit)
                        BIN: Binary feature values with \xff delimiters
                        (VASIM)
                        CSV: Comma-separated ASCII features with newlines
                        between vectors
  --merge               Combine all states from each tree that report the
                        same classification
  -d DIMENSIONS         Automata Folding with this many dimensions
  -v, --verbose         Verbose
```

5.1.4 Input Formats

Finite State Automata process an input stream by evaluating input symbols against character sets, setting state activity, and proceeding to the next state. Dif-

ferent simulators use different formats for input types. We currently support the following formats:

- Simple: This file is human-readable with feature values represented in their ASCII representations, and feature vectors separated by newlines. This input representation is used by our circuit simulator.

- Bin: This binary file format represents the input stream as a series of bytes, the same bytes that would be sent to an AP or FPGA with $0xFF$ bytes to delimit inputs. This file format is used by the spatial architectures as well as VASim and is most space-efficient of the available formats.

- CSV: This file format is Comma-separated-value with features separated by commas, and inputs separated by new-lines. This format is also supported by the circuit simulator.

5.1.5 Outputs

RFAutomata produces an automata ANML file. ANML is an XML-based automata representation created by Micron. There are tools that natively execute this format on the CPU (VASim[105]), Automata Processor[22](Native) and FPGA (REAPR[115]).

RFAutomata also produces an input symbol file, and other supporting files including a *Feature Range Lookup Table (FRLT)* for generating new inputs. The FRLT is used to convert raw input feature values into labels for automata processing. This can be done during feature extraction or as a separate pipeline stage on an FPGA.

5.1.6 Direct Hardware Support

RFAutomata supports GPU and ASIC hardware platforms by generating an intermediate representation that can be executed on these architectures.

GPU Automata Chains

We have developed an automata-based CUDA implementation that represents the independent automata chains in memory for execution on the GPU; one chain per thread. This program called RFAutomataGPU is a GPU accelerator for chain automata.

RFAutomata generates a GPU output format, shown below. The first line is the number of automata in the file. The second is the number of states per chain; the third the state at which the automata loop starts. Finally, the last header information contains the number of features used by the model. Then subsequent lines in the file start with the name of the automata and one character set per state.

```
19828
62
62
526
0t_0l_2r
[\x00-\x04\x05-\x09\x0A-\x0B\x0C-\x33\x34-\x47\x48-\x8C\x8D-\xC2\xC3-\xD6\...]
[\x00-\x03\x04-\x0A\x0B-\x0C\x0D-\x38\x39-\x76\x77-\xB2\xB3-\xD2\xD3-\xEA\...]
[\x00-\x03\x04-\x1A\x1B-\x1C\x1D-\x23\x24-\x62\x63-\x9D\x9E-\xC9\xCA-\xDB\...]
[\x00-\x03\x04-\x05\x06-\x1C\x1D-\x23\x24-\x45\x46-\x8F\x90-\xB6\xB7-\xE9...]
[\x00-\x04\x05-\x09\x0A-\x0B\x0C-\x20\x21-\x44\x45-\x6C\x6D-\xA0\xA1-\xE7...]
[\x00-\x03\x04-\x1A\x1B-\x1C\x1D-\x23\x24-\x5A\x5B-\x9D\x9E-\xC9\xCA-\xE9...]
...
0t_1l_8r
[\x00-\x04\x05-\x09\x0A-\x0B\x0C-\x33\x34-\x47\x48-\x8C\x8D-\xC2\xC3-\xD6...]
[\x00-\x03\x04-\x0A\x0B-\x0C\x0D-\x38\x39-\x76\x77-\xB2\xB3-\xD2\xD3-\xEA...]
[\x00-\x03\x04-\x1A\x1B-\x1C\x1D-\x23\x24-\x62\x63-\x9D\x9E-\xC9\xCA-\xDB...]
[\x00-\x03\x04-\x05\x06-\x1C\x1D-\x23\x24-\x45\x46-\x8F\x90-\xB6\xB7-\xE9...]
[\x00-\x04\x05-\x09\x0A-\x0B\x0C-\x20\x21-\x44\x45-\x6C\x6D-\xA0\xA1-\xE7...]
[\x00-\x03\x04-\x1A\x1B-\x1C\x1D-\x23\x24-\x5A\x5B-\x9D\x9E-\xC9\xCA-\xE9...]
```

```
[\x00-\x04\x05-\x19\x1A-\x1E\x1F-\x46\x47-\x48\x49-\x6C\x6D-\xB0\xB1-\xE7...]
...
```

Generic Circuits

We also support generic circuits. This works by representing the states in the automata as a series of simple upper- and lower-end comparisons that can be implemented in the target circuit with hardware comparators or other means of making the comparisons. Below is an example snippet from the output format from the $--circuits$ option. Each chain is represented with three separate lines: chain id, threshold ranges, and leaf value. This format is used by the work of Gonzalez et al.[32] in their asynchronous computing implementation.

```
0
0.0-22.0,0.0-256.0,0.0-0.5,0.0-256.0,0.0-256.0,0.0-256.0,0.0-256.0, ...
0

1
22.0-256.0,0.0-256.0,0.0-0.5,0.0-256.0,0.0-256.0,0.0-256.0,0.0-256.0, ...
3

2
0.0-256.0,0.0-256.0,0.0-0.5,0.0-256.0,0.0-0.5,0.0-256.0,0.0-256.0, ...
7

3
0.0-256.0,0.0-256.0,0.0-0.5,0.0-256.0,0.5-256.0,0.0-256.0,0.0-256.0, ...
3

...
```

5.2 REAPRpp

REAPRpp is an open-source synthesis tool for converting automata ANML files into hardware description language (HDL) files for deployment on FPGAs. We extend the work of Xie et al. and their REAPR[115] tool by including a parameterizable reporting architecture, debugging support, and support for targeting Amazon's AWS F1 instances. REAPRpp uses parameterized System Verilog modules to simplify deployment; it generates a REAPR header file that sets the parameters for reporting, debugging, and I/O.

REAPRpp performs the same HDL generation from an ANML automata description file as REAPR. Where REAPR generates VHDL automata files, REAPRpp generates a higher level System Verilog automata output files. One limitation with REAPR is that the I/O is statically configured in the generated modules. We use a dynamic approach by generating a Wrapper file around the generated automata modules. This separates automata logic from the I/O and allows us to parameterize both the reporting as well as the I/O.

To run the REAPR tool, run:

```
python3 reapr.py -a Model.anml
```

5.2.1 Reporting Architecture

REAPRpp includes support for the work of Jack et al. [102] and their automata reporting architecture. REAPR originally provided an output interface that mapped one wire to each reporting state in the automata. This is a significant limitation when targeting FPGAs and ASICs because IO is often a limiting factor. Our reporting architecture, as shown in Figure 5-4, serializes a large output reporting vector using Report Aggregators (RAGGs). Each RAGG receives a subset of the total report signals from the automata. When any RAGG receives at least one reporting signal, the Arbiter sends the contents of all active RAGGs off to the CPU.

We allow the user to provide either the number of RAGGs ($--numraggs$) or the width of the RAGGs ($--raggwidth$). This information is then used to

parameterize the reporting architecture modules to handle the reporting bits from the automata. Decision tree ensembles result in one inference from each tree in the ensemble, producing potentially hundreds of report bits per input. This is not a problem, because all reports occur in the same clock cycle, and the reports can be serialized and removed from the spatial architecture in time for the next set of bits, depending on the size of the input label vector. If there are more RAGGs that need to be serialized for reporting than their are input symbols, the automata are stalled by the Arbiter.

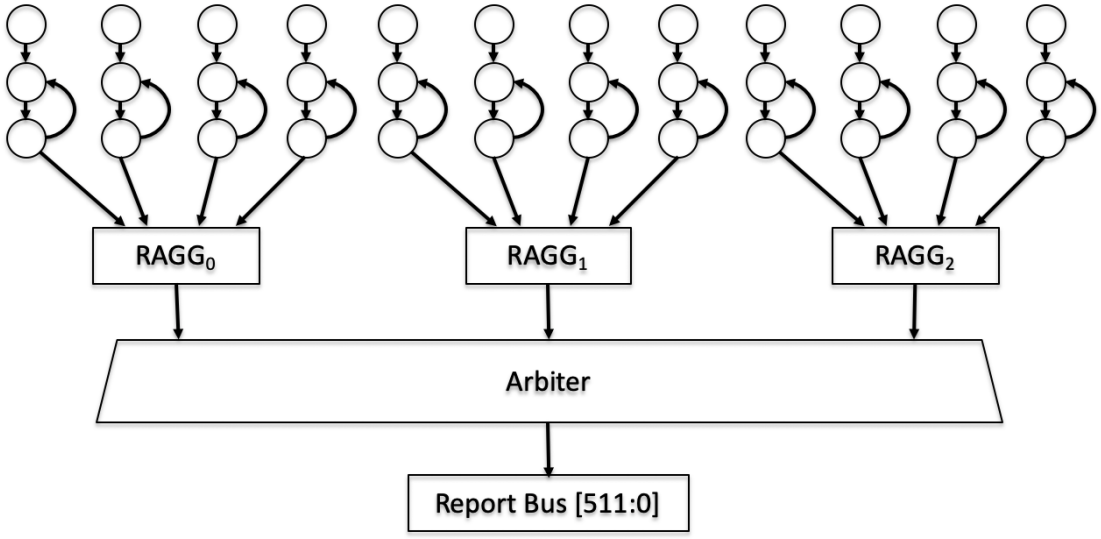


Figure 5-4: Reporting architecture for large automata designs[102].

5.2.2 Debugging Support

REAPRpp also includes the first example of an Automata Accelerator-based debugging architecture [16]. We have included a $--$*debug* option to REAPRpp that inserts Xilinx Virtual IO blocks[117] into the automata structure during RTL generation time. Once the design is deployed, the user can use Vivado to sample all of the STE state signals at runtime. This gives the user a snapshot of the state of all of the automata, providing a full picture of the automata configuration.

This debugging functionality comes at a price of hardware utilization, clock, and

power overhead. We were able to integrate the VIOs for all automata in the ANML-Zoo bechmark (which we introduce later in this chapter), maintaining an average of 81.7% of the baseline clock frequencies, a 2.82× and 6.09× LUT and FF overhead, respectively, and a 1.76× power overhead. We conducted a human study with over 60 participants and 20 buggy segments of code; our debugging information provided by REAPRpp's debugging functionality increased fault localization by 22%[16].

5.2.3 Amazon Web Services EC2-F1

Amazon's Elastic Compute Cloud (EC2) is a cloud computing service that allows users to rent virtual computers to run applications in the Cloud. Recently, Amazon has added instance types that include GPUs and FPGAs so that customers can take advantage of hardware acceleration. Amazon EC2 F1[1] instances include Xilinx FPGAs. Amazon also offers a variety of FPGA development instances that have FPGA development tools and licences pre-installed. These instances significantly reduce the complexity of accelerated application development and deployment, and eliminates the significant upfront monetary costs associated with development tool licenses and FPGA hardware. One f1.2xlarge instance offers 1 Xilinx FPGA with 64GB of on-chip DRAM, 8 virtual CPUs, and 122GB of host DRAM for just $1.65 per hour of use.

AWS provides an on-FPGA controller called the *Shell* that simplifies hardware and software development of applications. The shell is a statically programmed region of the FPGA that handles I/O tasks such as programming the dynamic region of the FPGA and implements a full AXI4 interface to the on-board DRAM.

One limitation of the Shell is that it limits the maximum frequency of its main clock to 250MHz, 125MHz, or 15.625MHz. An important metric for automata computing is the symbol processing rate, or the number of symbols that can be processed by the automata in a given time. For a standard one byte-per-cycle automata design (optimistically assuming a no-stall pipeline), a 250 MHz clock speed will yield a maximum throughput rate of 250 megasamples per second. Importantly, if the automata design cannot meet the shell's maximum 250MHz timing, by default the Shell will

use the next fastest supplied 125MHz clock: a 50% hit to performance. While the automata and shell can run at separate frequencies, we do not explore multi-clock domain designs. We leave this to future work.

Meeting Timing

Because the AWS Shell provides three significantly-spaced clock frequencies and the performance of automata computing is bound to the symbol rate, we were motivated to close our design within the highest 250MHz clock timing constraints. This presented a significant challenge to us for several reasons.

First, every STE (of the 10s to 100s of thousands of STEs) needed to receive all incoming data in the same clock cycle. This is particularly challenging as the input data stream is provided by a DDR RAM interface and would result in a massive fan-out. We managed to reduce the impact of the fanout on clock delay by capping a maximum fanout of 256. Vivado then duplicated these high-fanout paths across the chip.

The second significant challenge was dealing with long critical paths. Even with duplicated paths, the paths with input signals and paths back with output signals were very difficult to route meeting the tight timing constraints.

5.2.4 Recursively Grouping Automata

We introduce a balanced tree hierarchy to flat automata design in REAPRpp. Instead of relying on compiler tools to best optimize the logic across the full, flat design and place and route the resulting logic, we recursively partition automata subgraphs into groups of N, where N is the out-degree of our balanced tree structure.

We found that most application workloads were composed of many sub-graphs, and we used those subgraphs as the lowest unit of automata computing. We started by breaking apart the large graph into independent subgraphs with our technique illustrated in Algorithm 2. We do this by selecting a random seed state from the flat graph and sequentially traversing to outgoing as well as incoming states, adding them

all to the same subgraph list. We continue doing this until we have broken apart the flat graph into a large set of connected subgraphs.

```
Result: List of subgraphs
subgraphs = [];
temp_graph = [];
while flat_graph not empty do
    seed_state = flat_graph[0];
    state_stack = Stack();
    state_stack.push(seed_state);
    while state_stack not empty do
        state = state_stack.pop();
        temp_graph.append(state);
        flat_graph.pop(state);
        for outgoing_edge in state.outgoing do
            if outgoing_edge in flat_graph then
                state_stack.push(outgoing _edge);
            end
        end
        for incoming_edge in state.incoming do
            if incoming_edge in flat_graph then
                state_stack.push(incoming_edge);
            end
        end
    end
    subgraphs.append(temp_graph);
end
```

Algorithm 2: Flat graph to subgraphs

Then we combine these subgraphs by recursively grouping them into groups of N. Algorithm 3 shows how a balanced tree of degree *split_factor* is recursively built from the set of subgraphs. At each level, the algorithm generates N new hierarchical modules. We then continue to group the hierarchical modules recursively into groups of N until we've run out of groups.

Figure 5-5 shows the resulting balanced-tree hierarchy. The out-degree of our balanced tree is configurable, but in the diagram we show an out-degree of 2; for our experiments, we set it to 10. Additionally, at each split in the tree structure, we insert pipelining registers shown as red rectangles. From the input to the bottom-most (level 0) module, all symbol, reset, and output signal lines are registered the same

```
recursivesplit(self)
if len(self.subgraphs) < split_factor then
    return;
end
new_subgraphs = [subgraph for self.subgraphs.split(split_factor)];
new_reporting_states = [reporting_state for
 self.reporting_states.split(split_factor)];
for new_states in new_subgraphs do
    if len(new_states) < split_factor then
        split = False;
    else
        split = True;
    end
    for new_states, new_reports in zip(new_subgraphs, new_reporting_states)
     do
        new_node = Node(subgraphs=new_subgraphs,
         reporting_states=new_reports)
        if split then
            new_node.recursivesplit();
        end
        self.children.append(new_node);
    end
end
```

Algorithm 3: Recursively grouping automata

number of counts, ensuring that all automata graphs receive the same input data in the same clock cycle, and that all reporting output data makes it back to the I/O interface in the same clock cycle.

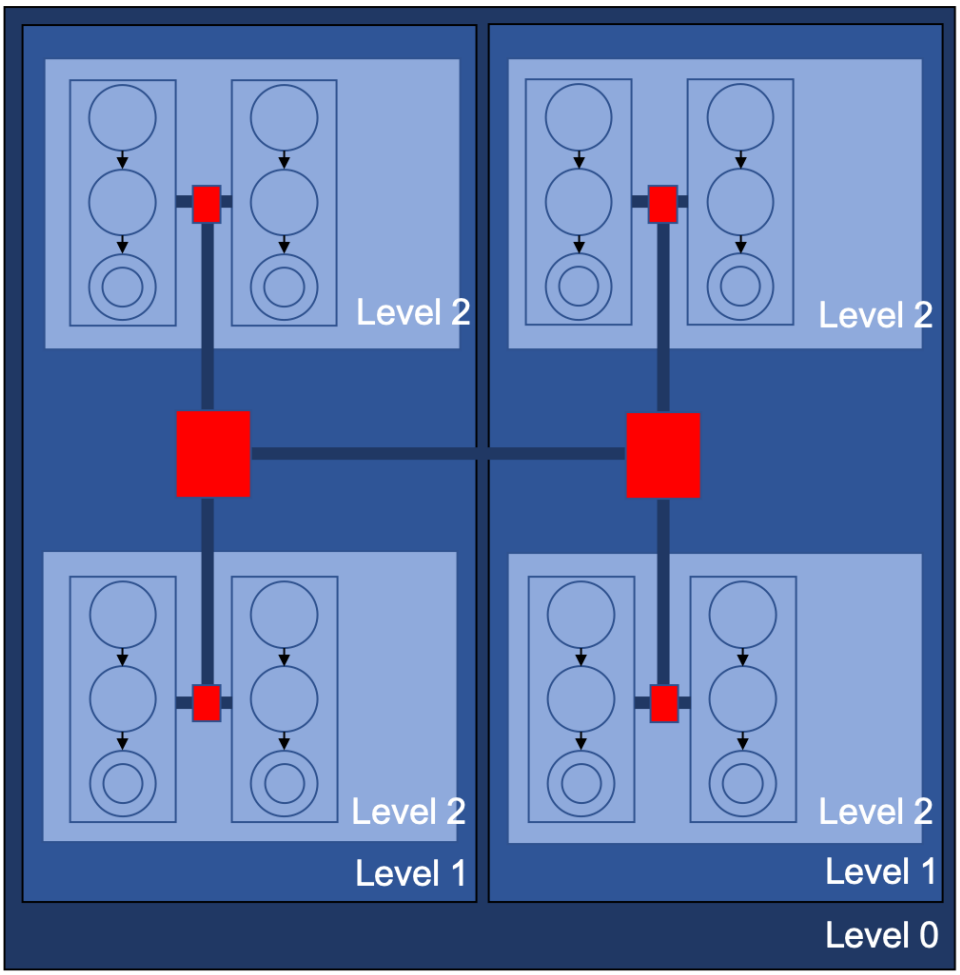


Figure 5-5: Recursively grouping automata into modules with preserved hierarchy.

Finally, we generate a separate Verilog module for each node in the balanced tree. The leaf nodes contain the Verilog automata structures, while the other branches contain pipeline registers for input and output signals. In order to enforce hierarchy and disable global optimizations across module boundaries, we use the following Vivado flag when instantiating child node modules: $(*keep\_hierarchy = "yes"*)$. This balanced H-Tree design with pipeline registers approach significantly reduced our compilation time by 50% and made larger designs over 50,000 STEs routable, but

came at the cost of 5% LUT overhead.

5.2.5 Additional REAPRpp Functionality

Below are additional parameters that we expose to the user:

```
usage: reapr.py [-h] -a AUTOMATA [-o OUTFILE] [-e ENTITY] [-w WRAPPER]
                [-d DEBUG] [--generate_vios] [--stes_per_probe STES_PER_PROBE]
                [--numraggs NUMRAGGS] [--raggwidth RAGGWIDTH]
                [--indexwidth INDEXWIDTH] [-m REPORTMAP]
                [--combine_bitcolumns] [--raggmap RAGGMAP] [--split SPLIT]
                [-v]

Generate an AWS F1-compatible FPGA accelerator for automata processing.

optional arguments:
  -h, --help            show this help message and exit
  -a AUTOMATA, --automata AUTOMATA
                        Input automata filename (ANML format).
  -o OUTFILE, --outfile OUTFILE
                        Output filename (defaults to <entity>.sv).
  -e ENTITY, --entity ENTITY
                        Automata entity name (defaults to 'automata').
  -w WRAPPER, --wrapper WRAPPER
                        Generate a wrapper around the automata
  -d DEBUG, --debug DEBUG
                        FOR DEBUGGING: Include debugging hardware and generate
                        xdc file for debugging cores
  --stes_per_probe STES_PER_PROBE
                        FOR DEBUGGING: Number of states sampled per probe
  --numraggs NUMRAGGS   If generating a wrapper, set the number of report
```

```
                        aggregators (RAGGs) used for reporting.
  --raggwidth RAGGWIDTH
                        If generating a wrapper, set the number of report bits
                        assigned to each report aggregator (RAGG).
  --indexwidth INDEXWIDTH
                        If generating a wrapper, set the number of metadata
                        bits assigned to each report aggregator (RAGG) for
                        counting input symbols.
  -m REPORTMAP, --reportmap REPORTMAP
                        Generate a report mapping file (element name to index)
  --combine_bitcolumns  Have STEs share equivalent bitcolumns
  --raggmap RAGGMAP     A ragg mapping file that maps regular expressions to RAGGs
  --split SPLIT         Split the automata into connected graphs and populate
                        ./tmp directory with separate modules
  -v                    Verbosity flag for details.
```

5.3 Benchmarks

The University of Virginia's Center for Automata Processing (CAP) made several significant contributions to the automata computing research community. Two of those contributions are the ANMLZoo[104] and later the AutomataZoo[106] benchmark suites. Up until their release, high-performance automata-processing engines were evaluated against a limited set of regular expression rulesets. Our research projects contributed a set of new automata-based applications to the benchmarks, and we contribute several Random Forest models as well as the tools to generate them.

5.3.1 ANMLZoo

ANMLZoo, named after Micron's ANML Automata markup language, was the first automata benchmark suite that included the new and growing diversity of automata applications. Up until it's release in 2016, benchmarks used for evaluating automata accelerators and tools were based on large sets of regular expressions [74, 44, 8], several of which were synthetic [4]. With the recent boom of automata applications, there was a need to address this new diversity of automata applications with matching benchmarks.

Micron's Automata Processor architecture, with the included software development kit and simulation tools, significantly reduced the effort required to prototype and develop automata-based pattern matching, reducing the overhead that this approach previous had over regular expression-based solutions. Being able to directly work with automata made reasoning about these abstract machines more concrete and stimulated new research in the field. Out of this explosion of automata research came several novel automata-accelerated applications including big data analysis[10], data mining[110], bioinformatics[75, 89], high energy particle physics[111], the machine learning research discussed in this dissertation, pseudo-random number generation and simulation[103], and natural language processing[122]. Many of these applications significantly differ from the previous works' regular expressions in structure and runtime behavior.

To that end, ANMLZoo became a standard for evaluating architectures and tools in the automata space. It is a benchmark suite of automata-based applications for evaluating software and hardware automata processing engines. It contains 14 different automata applications that represent four different classes: regular expressions, string scoring machines, programmable widgets, and synthetic automata. It also contains tool for evaluating these benchmarks on four different architectures: Intel i7-5820k, Intel's XeonPhi 3120, a Maxwell-based NVidia Titan X GPU, and Micron's Automata Processor.

We contributed one Random Forest (RF) model trained on the MNIST dataset to

the ANMLZoo benchmark suite. We reduced the number of features considered by the model to 300 from 784 and trained the RF on 15 decision trees. We then pruned the resulting automata chains to fit within the hardware limitations of a single Automata Processor chip. We also include 1MB and 10MB input stream files to serve as input to the automata benchmark.

5.3.2 AutomataZoo

We received feedback from the community indicating that ANMLZoo could be improved. A list of the drawbacks that we addressed with AutomataZoo[106] include:

- Promoting a now obsolete baseline: Micron's AP is not commercially available and built on an old design node. ANMLZoo benchmarks were designed around the capacity of one of the AP's chips, and this limited the design space of the benchmarks. Additionally, the benchmarks inherited the capacity and routing characteristics from the AP.
- Incomplete Benchmarks: In order to standardize to the capacity of one AP chip, many of the benchmarks were not complete applications, and were actually cut down to fit one chip. This made it impossible to make cross-algorithm performance comparisons.
- Most of the ANMLZoo benchmarks were a single design point and did not explore the spatial/temporal tradeoffs.

We address all of these concerns in AutomataZoo with 24 new benchmarks shown in Figure 5-6, where each is designed to be a real-world use case with multiple design points.

In order to give a more fair evaluation of automata across architectures, we demonstrated how to make full-kernel algorithm comparisons and we used the Random Forest benchmarks to make that comparison. We evaluated the full Random Forest models as automata on the CPU and FPGA versus a native Random Forest implementation on the CPU.

Benchmark	Domain	Input	States	Edges	Edges/ Node	Subgraph Count	Avg. Size	Std. Dev	Compressed States	Compr. factor	Size vs ANMLZoo	Active Set
Snort	Network Intrusion Detection	PCAP file	202,043	235,488	1.17	2,486	81.27	1.13	162,514	0.20x	4.71x	409.358
ClamAV	Virus Detection	Disk image	2,374,717	2,377,737	1.00	33,171	71.59	0.47	2,254,598	0.05x	53x	356.532
Protomata	Motif Search	Uniprot Database	24,103	24,031	1.00	1,309	18.41	1.13	22,289	0.08x	0.58x	712.884
Brill	Part of Speech Tagging	Brown Corpus	115,549	157,917	1.37	5,946	19.43	0.02	72,637	0.37x	2.76x	78.2558
Random Forest A	Machine Learning	Custom	248,000	248,000	1.00	8,000	31	0	240,001	0.03x	7.60x	862.504
Random Forest B	Machine Learning	Custom	248,000	248,000	1.00	8,000	31	0	240,001	0.03x	7.60x	1,043.18
Random Forest C	Machine Learning	Custom	992,000	992,000	1.00	16,000	62	0	976,001	0.02x	30.93x	2,334.97
Hamming 18x3	String Similarity	Random DNA	108,000	183,000	1.69	1,000	108	0	87,881	0.19x	7.81x	1,944.38
Hamming 22x5	String Similarity	Random DNA	192,000	347,000	1.81	1,000	192	0	168,936	0.12x	15.01x	6,324.49
Hamming 31x10	String Similarity	Random DNA	451,000	857,000	1.90	1,000	451	0	427,782	0.05x	38.01x	19,617.8
Levenshtein 19x3	String Similarity	Random DNA	109,000	445,000	4.08	1,000	109	0	91,007	0.17x	34.21x	4,528.69
Levenshtein 24x5	String Similarity	Random DNA	204,000	1,251,000	6.13	1,000	204	0	183,459	0.10x	68.97x	18,033.9
Levenshtein 37x10	String Similarity	Random DNA	557,000	6,221,000	11.17	1,000	557	0	530,981	0.05x	199.62x	85,866.1
Seq. Match 6w 6p	Ordered Pattern Counting	Custom	51,570	110,016	2.13	1,719	30	0	51,570	0x	0.51x	5,538.98
Seq. Match 6w 6p wC†	Ordered Pattern Counting	Custom	53,289	113,454	2.13	1,719	31	0	53,289	0x	0.53x	5,555.98
Seq. Match 6w 10p	Ordered Pattern Counting	Custom	85,950	185,652	2.16	1,719	50	0	85,950	0x	0.86x	5,465.23
Seq. Match 6w 10p wC†	Ordered Pattern Counting	Custom	87,669	189,090	2.16	1,719	51	0	87,669	0x	0.87x	5,497.23
Entity Resolution	Duplicate entry identification	100k names	413,352	641,136	1.55	10,000	41.34	0.11	309,475	0.25x	54.40x	57.5615
CRISPR CasOffinder	DNA pattern search	DNA	74,000	94,000	1.27	2,000	37	0	47,901	0.35x	NA	191.64
CRISPR CasOT	DNA pattern search	DNA	202,000	336,000	1.66	2,000	101	0	162,930	0.19x	NA	953.753
YARA	Malware pattern search	Malware files	1,047,528	1,025,863	0.98	23,530	44.52	0.22	482,233	1x	NA	579.739
YARA Wide	Malware pattern search	Malware files	115,246	112,910	0.98	2,620	43.99	0.20	82,837	0.28x	NA	123.964
File Carving	File metadata search	Multi-media files	2,663	156,601	58.81	9	295.89	93.30	699	0.74x	NA	15.6547
AP PRNG 4-sided	Pseudo-random number generation	Pseudo-random bytes	20,000	32,000	1.60	1,000	20	0	NA	NA	NA	4,500
AP PRNG 8-sided	Pseudo-random number generation	Pseudo-random bytes	72,000	128,000	1.78	1,000	72	0	NA	NA	NA	2,500

Figure 5-6: Automatazoo Benchmarks[106]

We used Intel's Hyerscan [31] automata engine with help from the MNCaRT [2] automata processing ecosystem. For an FPGA-based automata processing evaluation, we used Ted Xie's REAPR [115], and for native Random Forest computation on the CPU, we used single and multi-threaded version of Scikit-Learn [68]. All CPU results were measured on an Intel i7-5870k 6- core (12 logical core) server. Multi-threading experiments were conducted with 12 threads. FPGA results were obtained by placing and routing REAPR-generated automata on a Xilinx Kintex Ultrascale XCKU060 FPGA and multiplying the resulting maximum clock frequency by the input symbols required to drive the automata[106].

We normalized the results to the single-threaded Hyperscan performance shown in Table 5.2. Scikit-Learn achieved a 141.5× speedup in single-threaded performance over Hyperscan. This indicated that our automata approach was not a viable method to accelerated decision tree ensembles on von Neumann architectures because of the increased model size countered any improvement in access pattern. The FPGA-based REAPR automata achieved the best performance, but showed reduced performance improvements over the native Scikit-Learn implementation. These results highlight the usefulness of having complete application benchmark kernels to allow for useful

inter-architecture comparisons.

Hyperscan	Scikit Learn	Scikit Learn MT	REAPR FPGA
1×	141.5×	401.1×	817.9×

Table 5.2: Performance in kilo classifications/second of the Random Forest Automata, normalized to single-threaded Hyperscan[106]

We contributed three different Random Forest (RF) models all trained on the MNIST dataset. Unlike our ANMLZoo benchmark model, these models are complete and with interpretable results. This allows researchers to compare end-to-end performance between automata-based and non-automata-based accelerators and software engines.

We trained each RF model with 20 trees and varied the number of features and leaves per tree as shown in Table 5.3. Variant A and B differ by the number of features used to train the model as well as the number of features in the input stream. We chose 270 features for variant A, because we experimentally determined any increase beyond this number of features resulted in a negligible impact on accuracy of the model and only resulted in increased runtime. Variants B and C used 200 features to stay above a 90% accuracy.

Variants B and C differed by the maximum number of leaf nodes per tree, increasing the size of the tree model. With 200 features, increasing the maximum leaf count from 400 to 800 increased accuracy by almost a full percentage point, but it came at the cost of quadrupling the model size.

Variant	Features	Max Leaves	States	Accuracy	Runtime
A	270	400	248k	93.37	1.35×
B	200	400	248k	92.91	1.0×
C	200	800	992k	93.85	1.0×

Table 5.3: Random Forest benchmark variant trade-offs. Increasing the number of features increases accuracy but also increases runtime. Increasing the maximum number of leaves per tree increases accuracy, but increases automata size.[106]

We have released this new implementation to the ACM Sigarch community and we actively host the benchmark suite on github[91].

Chapter 6

Learn To Rank (LTR) Automata

In this Chapter, we apply the decision tree automata algorithms presented in Chapters 3 and 4, and the automata tools RFAutomata and REAPRpp presented in Chapter 5 to accelerate a state of the art Learn to Rank (LTR) document ranking algorithm used by Web search engines. We implement our LTR automata in the Amazon AWS Cloud on FPGAs and evaluate our approach against a high-performance CPU implementation to achieve significant speedups as well as reduction in power.

6.1 Learn To Rank Document Ranking

Learning to Rank (LtR)[50] is a supervised machine learning task used as a core component of information retrieval, natural language processing, and data mining systems. Given a large set of documents and a user query, LtR models learn to assign a score to each document in a candidate subset of the documents. These scores are based on each document's relevance to a user's input query. The scores are then used to return the most relevant documents to the user's query in ranked order. Although useful for other applications, we will consider LtR techniques for information retrieval[34] like those used by Web search engines.

LtR models are computationally expensive to use on large sets of documents. The model evaluates a score for the user's input query against each document in the document set. This is done by first constructing a feature vector with hundreds of

features for each (query, document) pair and computing a relevance score from that feature vector. To reduce the time and power costs of each LtR query, contemporary information retrieval systems use a two-stage processing pipeline[20] as shown in Figure 6-1.

In the first stage of the pipeine, a simple base ranking function is used to return the top-N results of a large database of documents using a less computationally intensive method like Okapi Best Matching 25 (BM25)[100]. BM25 is a bag-of-words retrieval function that assigns a score to a document based on the frequency of query terms in the document.

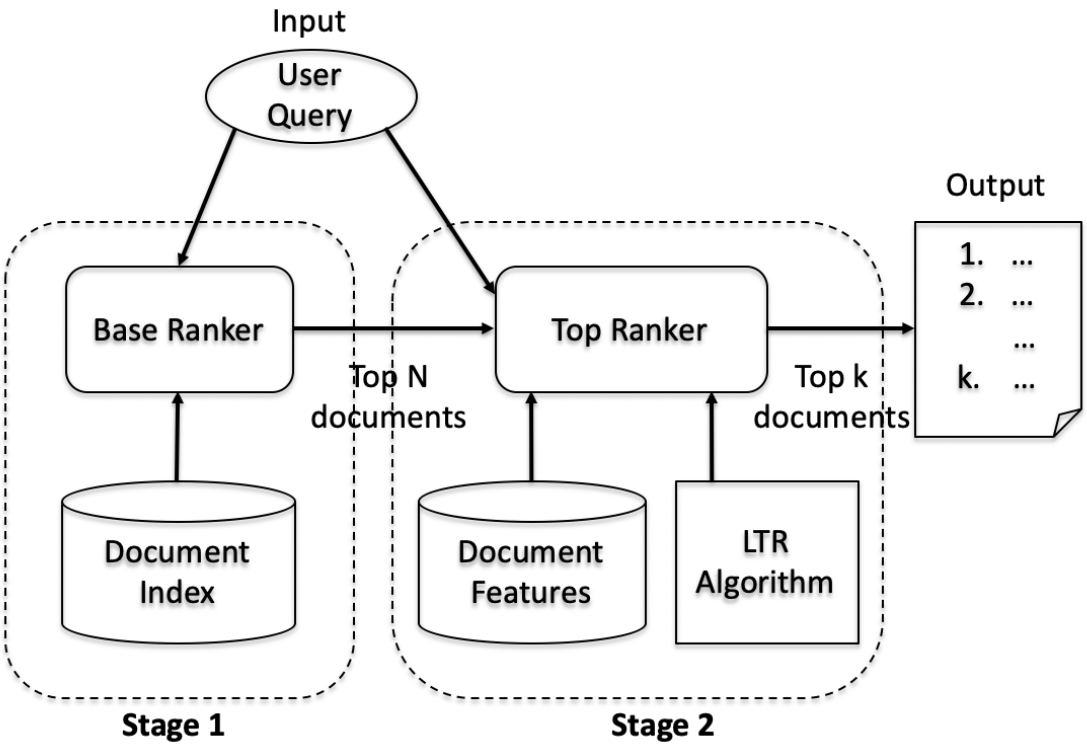


Figure 6-1: The two-stage LTR pipeline.

A high-precision LtR scorer is then used in the 2nd stage. Here the LtR scorer re-scores and ranks the candidate set of documents to achieve higher precision. The second stage returns the top-K results ranked in order where $K << N$.

LtR models that use additive ensembles of regression trees have proven very effective for ranking Web pages returned by search engines[51, 40, 52]. Quality is im-

portant, but so is efficiency and throughput when considering the number of queries search engines process each day. It is this second stage that we seek to accelerate with decision tree automata.

6.1.1 Training with LtR Models

Several modern LtR models are based on additive ensembles of Regression trees that are trained with the Gradient-Boosted Regression Tree (GBRT)[29] or LambdaMart[114] algorithms. These algorithms produce ensembles of 100s to 1000s of decision trees that all need to be traversed to come up with a single score for a web query against all of the k documents.

LtR models are ML models and have a training and testing phase. In the training phase, a large training dataset is used to generate the decision tree ensemble. This training dataset contains feature vectors extracted from (query, document) pairs with their associated relevance scores. Example features included in the extracted featured vector include BM25 [72] and PageRank [67] scores.

Normalized Discounted Cumulative Gain

Traditional classification and regression algorithms compute an inference on a single input data point. For this reason, metrics like least square error and accuracy are relevant, as they assess the difference between an inferred value and a ground truth.

LTR models are different in that they compute inferences on a list of items, learning a function to assign a score to each item such that their ordering is optimal. For this reason the score, itself, is not of importance; only the relative scores are. Therefore to measure the quality of an LtR model, it is necessary to measure the quality of an ordering of documents.

The LtR LambdaMART boosted regression tree model is trained to maximize the Normalized Discounted Cumulative Gain (nDCG)[39]. The nDCG is a measure of the quality of a ranking of documents and is commonly used to measure the effectiveness

of search engine algorithms. It calculates a discounted cumulative gain as the sum of the relevance values of the documents in a ranked list, and penalizes highly relevant documents that appear lower in the list of results.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

It then divides the discounted cumulative gain by the ideally-ranked discounted cumulative gain to achieve a normalized value in the range $[0, 1]$:

$$IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

$$nDCG_{10} = \frac{DCG_{10}}{IDCG_{10}}$$

6.2 QuickRank: High Performance LtR on CPUs

QuickRank[14] is a public-domain, C++ framework for evaluating the performance of various LtR models. It supports several LtR models including Gradient Boosted Regression Trees (GBRT)[29], Lambda-Mart[114], and Oblivious Lambda-Mart[84]. QuickRank provides learning algorithms for each of the models and generates optimized memory data structures to maximize scoring performance. It does this by translating learnt models into an efficient C++ source code that is recompiled into a ranker. This framework also includes timing support to give details about performance.

6.2.1 QuickScore

QuickRank uses QuickScorer[51] to efficiently compute the decision tree traversals. It does this be converting decision trees into bit vectors and uses nodemasks and bit-wise operations to interleave processing of regression trees for out-of-order computation. QuickScorer achieves significant performance increases (up to 6.5×) over previous boosted regression tree-based LTR models, establishing the state of the

art solution for LtR implementations on the CPU. To the best of our knowledge, it is the highest performing Boosted Regression Tree inference engine available for CPUs.

6.3 Experimental Setup

6.3.1 Microsoft Learning to Rank Dataset

In order to evaluate our automata-based decision tree approach, we use the first fold of Microsoft's LTR (MSLR) Bing dataset[70]. This dataset contains training, validation, and testing datasets. Each row in the dataset contains a relevance label from 0 (irrelevant) to 4 (perfectly relevant), followed by a query id, and 136 features extracted from the query-url pair. These features include, among others, the following information about the pair:

- The occurrence of the query term in the body and anchor of the URL
- PageRank, SiteRank, QualityScore, and length of URL
- URL dwell time, URL click count

Below is an example of a single entry in the MSLR dataset. Notice how the first item is the relevance label, the second the query ID, and the remaining are features in the form $< featureindex >:< featurevalue >$.

```
2 qid:13 1:2 2:0 3:2 4:1 5:2 6:1 7:0 8:1 9:0.50000 10:1 11:31 12:0 13:11
14:7 15:49 16:6.553125 17:15.011174 18:12.950828 19:14.369216 20:6.550869
21:4 22:0 23:2 24:1 25:7 26:2 27:0 28:1 29:0 30:3 31:2 32:0 33:1 34:1 35:4
36:2 37:0 38:1 39:0.50000 40:3.50000 41:0 42:0 43:0 44:0.25000 45:0.25000
46:0.129032 47:0 48:0.181818 49:0.142857 50:0.142857 51:0.064516 52:0
53:0.090909 54:0 55:0.061224 56:0.064516 57:0 58:0.090909 59:0.142857
60:0.081633 61:0.064516 62:0 63:0.090909 64:0.071429 65:0.071429 66:0
67:0 68:0 69:0.005102 70:0.000104 71:13.106251 72:0 73:12.950828 74:6.829093
75:22.821554 76:6.340183 77:0 78:6.129836 79:0 80:10.145764 81:6.766068 82:0
```

```
83:6.820992 84:6.829093 85:12.67579 86:6.553125 87:0 88:6.475414 89:3.414546
90:11.410777 91:0.045344 92:0 93:0.119424 94:11.659127 95:1.600258 96:1 97:0
98:1 99:0 100:1 101:1 102:0 103:1 104:0.671329 105:0.989811 106:17.818264 107:0
108:10.183562 109:7.633816 110:19.436549 111:-6.340431 112:-12.071142
113:-7.191141 114:-13.131176 115:-5.755162 116:-13.631532 117:-16.095443
118:-14.367199 119:-16.975368 120:-12.63974 121:-5.692009 122:-12.91985
123:-5.00585 124:-13.980776 125:-5.509102 126:2 127:35 128:1 129:0 130:266
131:25070 132:28 133:7 134:0 135:0 136:0
```

6.3.2 Training the LTR Model

We used the default arguments provided by QuickRank to train our Lambdamart model on the first Fold of Microsoft's MSLR dataset. We achieved a maximum 0.4545 NDCG@10 score with 800 trees as shown in Figure 6-2. We experimentally determined that our model saturated its NDCG score at 800 trees by allowing QuickRank to continue training the Lambdamart model until there was no increase in NDCG@10 score.

```
./quicklearn --algo LAMBDAMART --train ~/data/mslr/Fold1/train.txt
--valid ~/data/mslr/Fold1/vali.txt --train-metric NDCG --train-cutoff 10
--model-out lambdamart-model.xml
```

We then trained eight different models with between 100 and 800 trees. For each of the models, we used QuickRank's efficient scoring with CONDOP optimization to maximize our inference performance. This optimizer produced a new *model.cc* file that we compile and run to evaluate scoring inference.

```
./quicklearn --model-file lambdamart-model.xml --code-file model.cc
--generator condop
```

6.3.3 Evaluating LtR on the CPU

We then executed the optimized CPU models on an Intel i7-7700K CPU running at 4.20GHz with 4 cores and 8 virtual cores, 64GB of DDR4 RAM running at 2133MHz, and running Ubuntu Linux 16.04.6 LTS. For each model, we determined the nDCG@10 score as well as the throughput achieved by the CPU. As shown by the blue bars in Figure 6-2, we saw a significant decrease in throughput as the number of trees in the ensemble increase.

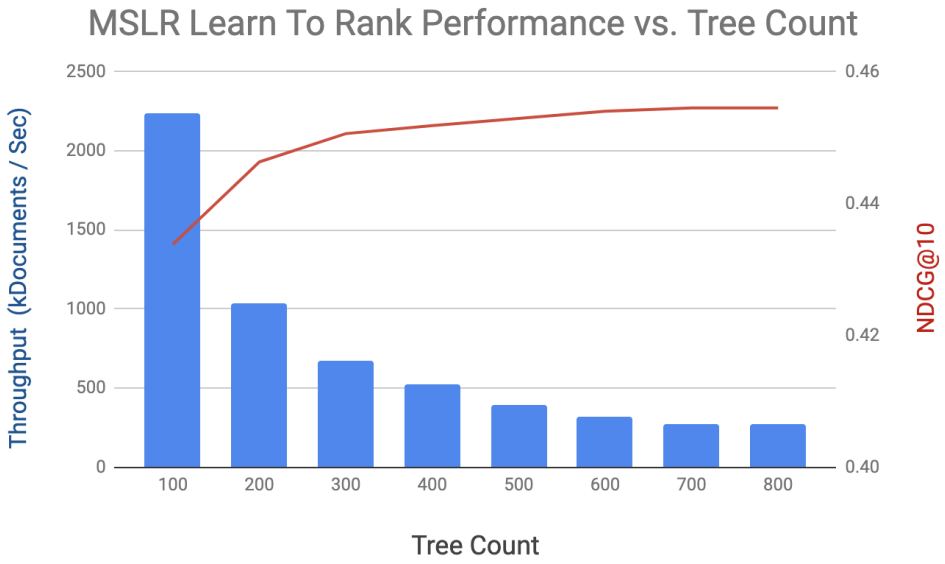


Figure 6-2: LTR throughput and NDCG@10 vs. number of trees.

6.3.4 Evaluating LtR on the Automata

We use RFAutomata to generate automata by converting the trained QuickRank models into ANML files. We then use REAPRpp to generate HDL by converting the ANML files into a set of System Verilog and header files. Finally, we compile the HDL files into a bitstream that we deploy on Amazon's AWS F1 [1]. We do this for all eight models trained by QuickRank. Table 6.1 shows the time it takes to generate the automata, generate the HDL, and compile the HDL for the FPGA on the smallest and largest models. The majority of the pre-computing time was spent on compiling the bitstream, with automata and RTL generation taking less than 1%

of the pre-computing time.

Trees	Generate Automata	Generate HDL	Compile for FPGA
100	0.01 min	0.87 min	91 min
800	0.22 min	9.58 min	2034 min

Table 6.1: Runtimes for converting ML models into automata, automata into HDL, and compiling the HDL into an FPGA bitstream.

We then evaluate our model's throughput against the CPU throughput as shown in Figure 6-3. We obtained approximately an 8× increase in throughput on the FPGA. When comparing our power results of our FPGA implementation as shown in Figure 6-4 to the approximated 90 Watt TDP of our CPU, we utilize between 41% and 48% of the CPU power on the FPGA with Vivado-calculated power measurements between 37 and 43 Watts.

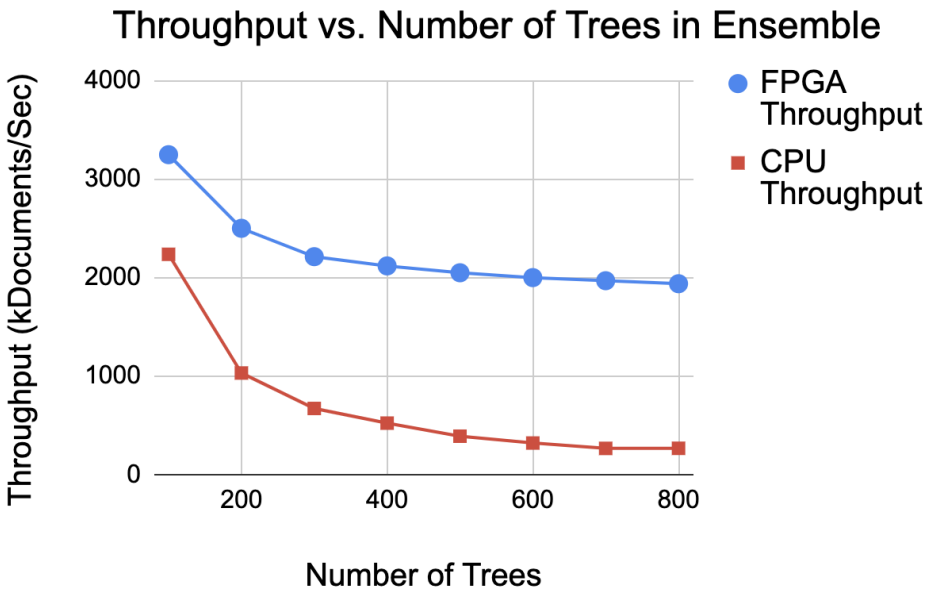


Figure 6-3: CPU throughput vs. FPGA throughput as a function of the number of trees.

6.3.5 Hardware Utilization

Figure 6-5 shows the hardware utilization of the AWS F1 FPGA as a function of the size of the LtR model. The 800 tree model used only 41.5% of the FPGA's

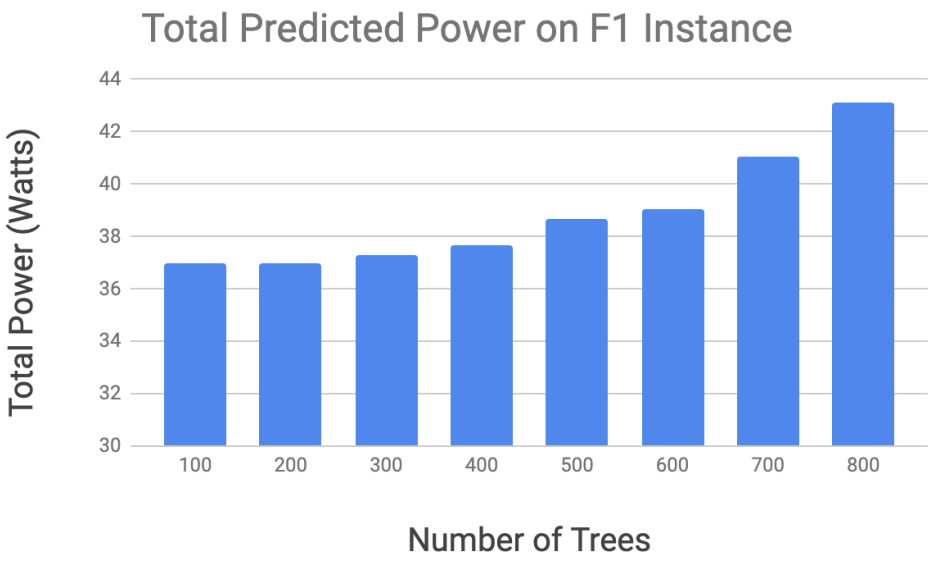


Figure 6-4: Total F1 power as a function of the number of trees.

resources, leaving a significant amount of usable space for multi-streaming. In our evaluation, we only used one of the four DRAM channels, leaving room for additional multi-streaming throughput.

Figure 6-6 shows the FPGA layout generated by Vivado. The orange components shown on the FPGA are the Shell components. The Shell is provided by AWS and implements PCIe, DRAM, DMA, interrupts and other external peripheral logic. The shell comes at a heavy price, costing approximately 20% of the FPGA's resources.

6.4 Implementation Challenges and Solutions

During the process of implementing our automata LtR models on AWS-F1, we encountered several complications as we implemented larger models. Our initial approach of one large flat design required long compilation times and we could not close on the intended 250MHz timing constraints.

6.4.1 Congestion

We encountered congestion issues that required that we adapt REAPRpp to improve the routability of larger automata designs. To reduce the critical path size, we

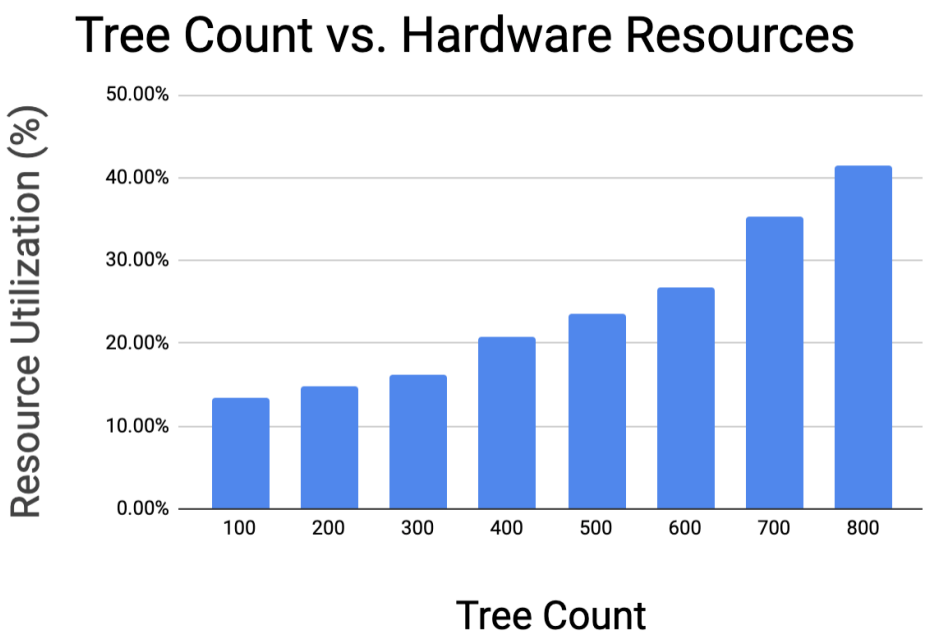


Figure 6-5: Percentage of hardware resourced utilized by the LtR model as a function of the number of trees in the ensemble.

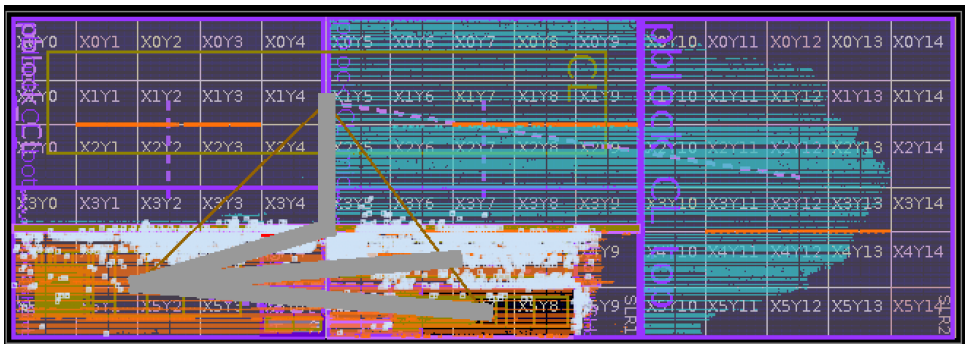


Figure 6-6: Diagram of the F1 FPGA utilization for our 800-tree LtR model; the orange components are the Shell, the blue the automata, and the grey show the wide input signals distributed to the STEs.

sub-moduled all subgraphs in our flat automata representation and enforced an artificial hiearchy by generating an H-Tree structure as described in Section 5.2.4. Each of the terminating nodes represented one automata from the ensemble. By inserting pipelining registers at each stage of the hierarchy, we could meet timing.

Using our recursive hierarchy approach, we were able to close the 250MHz timing for all of the 100-800 decision tree models, whereas without we could not for ensembles larger than 500 trees. Although the underlying functionality is preserved, by providing

a hint to the synthesis tools about the relationship between automata simplified the complexity of routing the design. Also the introduction of the pipeline registers increase the startup costs to fill the pipeline, but this approach reduced the critical path lengths of our design.

6.4.2 Compilation Time

One limitation of using the FPGA over other hardware platforms is the significant synthesis and place-and-route time. For our implementation, compile times ranged from 90 minutes for a smaller 100 tree ensemble to just under 34 hours for the 800 tree implementation! As shown in Figure 6-7, utilizing the H-Tree reduced our compilation time by up to 50% by significantly reducing the amount of time the compilation tools were optimizing the logic!

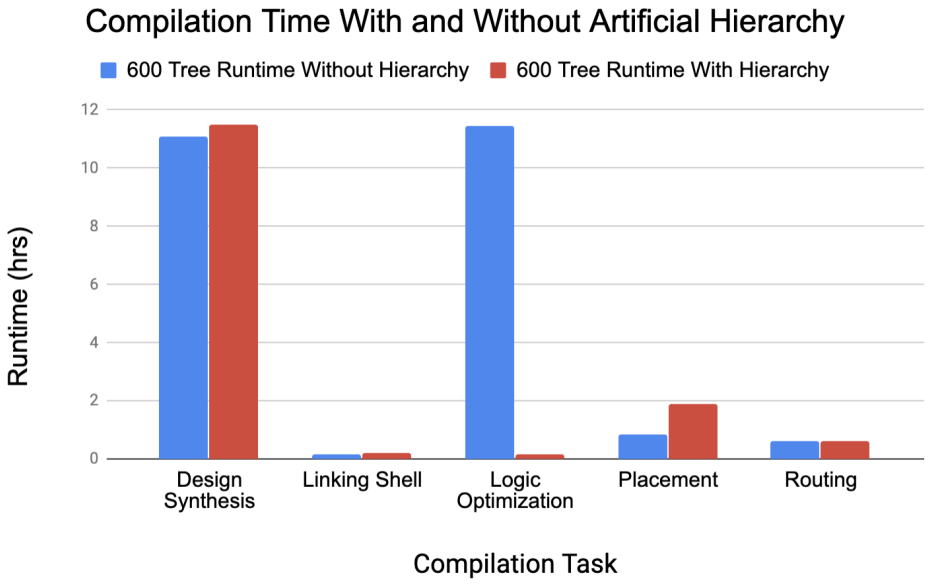


Figure 6-7: Time required to synthesize and place-and-route the 600-tree LtR design with and without the H-Tree hierarchy. *The design without H-Tree failed to meet 250MHz timing constraints.

Chapter 7

Conclusions and Future Work

7.1 Summary

Machine learning models based on a plurality of decision trees are versatile and high performing across a significant range of application and research domains. Their simplicity with few hyper-parameters to tune, fast training, and flexibility make them popular for prototyping. Their interpretability also provides valuable insight to the way they make their decisions.

With the increasing amount of data sourced from IoT devices, smart phones, and medical devices, accelerating decision tree inference rates as well as increasing their efficiency is critically important. Also, with the introduction of new layered decision tree models that obtain neural network-like performance, there is a need to increase the throughput that large decision tree models can sustain.

Unfortunately, accelerating the inference rate and efficiency of decision tree models presents a variety of technical challenges. Decision trees exhibit a memory access pattern with low locality that results in memory-bound von Neumann implementations. Execution divergence while traversing the many different paths of all of the decision trees stifles efficient parallel execution with SIMD architectures. Finally, existing work on accelerating these models on spatial architectures require buffering and significant hardware costs for memory management and floating-point comparisons.

The Automata Processor and FPGAs are non-von Neumann spatial processor ar-

chitectures that can efficiently compute thousands of small state machine automata in parallel on an input stream of data. We develop algorithms and tools for accelerating decision tree ensembles on these spatial architectures by converting them into spatial automata. In order to accomplish this transformation from decision tree ensemble to independently-computable state machines, we had to solve several complex challenges.

Firstly, we had to convert continuous numerical feature values into symbols from a discrete address space. We solved this problem by introducing a pipelined labeling technique that converted between these formats without any loss of fidelity to the original model. Secondly, we had to convert a local, data-dependent memory access pattern to a global, sequential, streaming access pattern for all automata on the spatial architecture. We solved this problem by fundamentally restructuring decision tree traversal into the task of filtering an input stream. Finally, our automata representation came at a significant spatial overhead. We reduce the spatial resources required to fit the large models onto existing architectures by introducing two compaction techniques called One-Hot and Two-Hot Automata Folding.

We developed two open source research tools: RFAutomata and REAPRpp. RFAutomata allows researchers and developers to convert between their existing decision tree ensemble models and our automata representation, catalyzing automata-based machine learning acceleration. RFAutomata also allows researchers to explore spatial and temporal design tradeoffs and evaluate their models on a variety of automata engines. REAPRpp is an enhanced synthesis tool for converting this automata representation into RTL for FPGAs.

Finally, we prove our hypothesis by accelerating several decision tree ensemble applications on spatial architectures by representing the constituent decision trees with finite state automata. One application we focused on was Learn-to-Rank document ranking. We targeted cloud-based FPGAs and achieve significant performance improvements and power reduction.

7.2 Impact

The work presented in this dissertation has been distributed to researchers and industry by way of several publications at workshops and conferences, as well as having been demonstrated at a research annual review meeting. To the best of our knowledge, we are the first researchers to demonstrate an end-to-end application on Micron's Automata Processor. We demonstrated a Random Forest-based character recognition application with Brill rule tagging at the Center for Future Architecture Research (CFAR) 2016 meeting at the University of Michigan. Our automata decision tree work was also accepted twice (2018 and 2019) at the competitive GOMACTech conference, a yearly conference established for the review of developments in microcircuit application with government applications.

In addition, our techniques are currently being patented and the open-source tools presented have been used by other researchers in other research projects. Finally, our tools have have achieved industry penetration and expect future work to expand adoption.

7.2.1 Automata Tools

Several research projects utilize our automata design tools including a current research project in asychronous, stochastic computing, using the automata representation of the decision tree ensembles to do machine learning inference at very low power, on a stream of values close to the sensor[32]. Our automata debugging paper[16] presented at the Architectural Support for Programming Languages and Operating Systems (ASPLOS) conference also utilized the debugging options built into REAPRpp.

7.2.2 Automata Benchmarks

In Chapter 5 we described the ANMLZoo and AutomataZoo benchmark suites with 28 and 2 citation counts as of 6 June 2019, respectively. These benchmarks are, to our best knowledge, the most diverse automata benchmarks currently available to

researchers, providing a variety of applications that traditional automata benchmarks do not.

7.2.3 Industry Penetration

The algorithms presented in Chapter 3 were also submitted and are currently being reviewed to be patented, under the ownership of Micron. The patent has a pending status[119] as of July 2019. In addition, we worked with an employee of Xilinx on implementing a Random Forest model, indicating some level of industry adoption.

7.3 Future Architecture Research

This dissertation introduces several novel techniques for transforming decision tree ensembles into state machines that can process an input stream with high parallelism. We have shown their implementation on CPUs, GPUs, FPGAs, Micron's Automata processor, and on an ASIC with stochastic computing logic. Future work could explore targeting additional architectures including MIMD architectures like Adapteva's Parallella[63]. One approach could utilize wave-style processing for automata computing across an array of lightweight processor cores.

7.3.1 Automata Overlays

Our approach has focused on implementing decision trees directly in hardware by mapping the State Transition Elements directly in the LUTs and connecting the STEs directly. Although this process allows for very lightweight design, it requires expensive synthesis and place-and-route every time a new design is implemented. We have explored techniques for reducing the impact of this overhead in Chapter 5, but there is room for additional work to further reduce compilation overhead.

Another approach is to use an overlay architecture. Future work could make a time/space tradeoff by using an Overlay architecture, something similar to the AP,

where the STE character sets and connectivity can be reconfigured much faster at the cost of additional power and spatial resources. This architecture would be useful for applications where rapid reconfigurability is important, a metric that this dissertation did not focus on.

7.3.2 Automata Machine Learning

We demonstrated in Chapter 3 that decision tree automata essentially map each of the learned partitions to a filter state machine that recognizes an input feature vector that maps to that partition. Instead of training decision trees and deconstructing them, future work could explore directly generating automata from the training data. With the development of flexible automata overlays, this process could be done in-situ on the spatial architecture.

This approach could also allow for continual learning, where as new training data is added, the automata assigned to partitions could adjust the ranges in each feature dimension to account for new statistical changes. Instead of rebuilding the entire ensemble, each of the automata would keep track of their own histograms, and as new data was added to their random subset and the histogram statistics changed, the automata could update their thresholds to accommodate the change. This would also require that the feature table be flexible in adding new feature ranges.

One approach for implementing in-situ decision tree automata learning could use a hard or soft core CPU and DRAM to handle the update computations and modify the BRAMs as new data arrives. We hypothesize that this approach could be very useful for space and automotive applications where reliability is a primary concern, for IOT devices where power is a primary concern, and for cybersecurity applications where rapid updates are important when responding to events.

7.3.3 Exploring Additional Machine Learning Models

We believe that decision trees and decision tree ensembles are not the only models that can be accelerated with our automata approach. We propose future research into

utilizing the techniques that we presented for other models including Support Vector Machines (SVMs) and binarized neural networks.

Bibliography

[1] EC Amazon. F1 instances: Run custom fpgas in the aws cloud, 2017. *URL https://aws. amazon. com/ec2/instance-types/f1*.

[2] Kevin Angstadt, Jack Wadden, Vinh Dang, Ted Xie, Dan Kramp, Westley Weimer, Mircea Stan, and Kevin Skadron. Mncart: An open-source, multi-architecture automata-processing research and execution ecosystem. *IEEE Computer Architecture Letters*, 17(1):84–87, 2017.

[3] Nima Asadi, Jimmy Lin, and Arjen P. de Vries. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):1–1, 2014.

[4] Kubilay Atasu, Florian Doerfler, Jan van Lunteren, and Christoph Hagleitner. Hardware-accelerated regular expression matching with overlap handling on ibm poweren processor. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1254–1265. IEEE, 2013.

[5] Susan Athey, Julie Tibshirani, Stefan Wager, et al. Generalized random forests. *The Annals of Statistics*, 47(2):1148–1178, 2019.

[6] Shuang Bai. Growing random forest on deep convolutional neural networks for scene categorization. *Expert Systems with Applications*, 71:279–287, 2017.

[7] Michela Becchi. *Data structures, algorithms and architectures for efficient regular expression evaluation*. Washington University in St. Louis, 2009.

[8] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2009.

[9] Simone Bendazzoli, Irene Brusini, Peter Damberg, Örjan Smedby, Leif Andersson, and Chunliang Wang. Automatic rat brain segmentation from mri using statistical shape models and random forest. In *Medical Imaging 2019: Image Processing*, volume 10949, page 109492O. International Society for Optics and Photonics, 2019.

[10] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. Entity resolution acceleration using micron's automata processor. *Proceedings of Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[11] Leo Breiman. Using adaptive bagging to debias regressions. Technical report, Technical Report 547, Statistics Dept. UCB, 1999.

[12] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[13] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*, pages 737–744, 1994.

[14] Gabriele Capannini, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, and Nicola Tonellotto. Quality versus efficiency in document scoring with learning-to-rank models. *Information Processing & Management*, 52(6):1161–1177, 2016.

[15] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: Nfa pattern matching on gpgpu devices. *ACM SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[16] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[17] Mauricio Castro-Franco, José Luis Costa, Nahuel Peralta, and Virginia Aparicio. Prediction of soil properties at farm scale using a model-based soil sampling scheme and random forest. *Soil science*, 180(2):74–85, 2015.

[18] Marie Chavent, Robin Genuer, and Jerome Saracco. Combining clustering of variables and feature selection using random forests. *Communications in Statistics-Simulation and Computation*, pages 1–20, 2019.

[19] Xing Chen, Li Huang, Di Xie, and Qi Zhao. Egbmmda: extreme gradient boosting machine for mirna-disease association prediction. *Cell death & disease*, 9(1):3, 2018.

[20] Van Dang, Michael Bendersky, and W Bruce Croft. Two-stage learning to rank for information retrieval. In *European Conference on Information Retrieval*, pages 423–434. Springer, 2013.

[21] Ana de Castro, Jorge Torres-Sánchez, Jose Peña, Francisco Jiménez-Brenes, Ovidiu Csillik, and Francisca López-Granados. An automatic random forest-obia algorithm for early weed mapping between and within crop rows using uav imagery. *Remote Sensing*, 10(2):285, 2018.

[22] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[23] Nejdet Dogru and Abdulhamit Subasi. Traffic accident detection using random forest classifier. In *2018 15th Learning and Technology Conference (L&T)*, pages 40–45. IEEE, 2018.

[24] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[25] Ji Feng and Zhi-Hua Zhou. Autoencoder by forest. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[26] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[27] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[28] Brady Fowler, Monica Rajendiran, Timothy Schroeder, Nicholas Bergh, Abigail Flower, and Hyojung Kang. Predicting patient revisits at the university of virginia health system emergency department. In *2017 Systems and Information Engineering Design Symposium (SIEDS)*, pages 253–258. IEEE, 2017.

[29] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[30] Jerome H Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002.

[31] Geoff Langdale. HyperScan in Suricata: State of the Union. 2016.

[32] Patricia Gonzalez, Tommy Tracy II, Xinfei Guo, Marzieh Lenjani, and Mircea R. Stan. Towards low-power machine learning using asynchronous computing with streams. Submitted for publication.

[33] Stuart K Grange, David C Carslaw, Alastair C Lewis, Eirini Boleti, and Christoph Hueglin. Random forest meteorological normalisation models for swiss pm 10 trend analysis. *Atmospheric Chemistry and Physics*, 18(9):6223–6239, 2018.

[34] LI Hang. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems*, 94(10):1854–1862, 2011.

[35] Satoshi Hara and Kohei Hayashi. Making tree ensembles interpretable. *arXiv preprint arXiv:1606.05390*, 2016.

[36] Abián Hernández, Himar Fabelo, Samuel Ortega, Abelardo Báez, Gustavo M Callicó, and Roberto Sarmiento. Random forest training stage acceleration using graphics processing units. In *2017 32nd Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2017.

[37] Tommy Tracy II. Rfautomata. https://github.com/hplp/rfautomata, 2018.

[38] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[39] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[40] Shiyu Ji, Jinjin Shao, Daniel Agun, and Tao Yang. Privacy-aware ranking with tree ensembles on the cloud. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 315–324. ACM, 2018.

[41] Amod Jog, Aaron Carass, Snehashis Roy, Dzung L Pham, and Jerry L Prince. Random forest regression for magnetic resonance image synthesis. *Medical image analysis*, 35:475–488, 2017.

[42] Rob Kassel. Ocr dataset, 2013.

[43] Sangwon Kim, Mira Jeong, Deokwoo Lee, and Byoung Chul Ko. Deep coupling of random ferns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 5–8, 2019.

[44] Tomasz Kojm. Clamav, 2004.

[45] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[46] Yann LeCun and Corinna Cortes. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

[47] Yeon-sup Lim, Hyun-chul Kim, Jiwoong Jeong, Chong-kwon Kim, Ted Taekyoung Kwon, and Yanghee Choi. Internet traffic classification demystified: on the sources of the discriminative power. In *Proceedings of the 6th International COnference*, page 9. ACM, 2010.

[48] Zhe Lin, Sharad Sinha, and Wei 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)*, pages 172–180. IEEE, 2019.

[49] Fei Tony Liu, Kai Ming Ting, Yang Yu, and Zhi-Hua Zhou. Spectrum of variable-random trees. *Journal of Artificial Intelligence Research*, 32:355–384, 2008.

[50] Tie-Yan Liu et al. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.

[51] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, pages 73–82, New York, NY, USA, 2015. ACM.

[52] Claudio Lucchese, Franco Maria Nardini, Raffaele Perego, Salvatore Orlando, and Salvatore Trani. Selective gradient boosting for effective learning to rank. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 155–164. ACM, 2018.

[53] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pages 142–150. Association for Computational Linguistics, 2011.

[54] Diego Marron, Albert Bifet, and Gianmarco De Francisci Morales. Random forests of very fast decision trees on gpu for mining evolving big data streams. In *ECAI*, volume 14, pages 615–620, 2014.

[55] Kevin Miller, Chris Hettinger, Jeffrey Humpherys, Tyler Jarvis, and David Kartchner. Forward thinking: Building deep random forests. *arXiv preprint arXiv:1705.07366*, 2017.

[56] Tom Mitchell, Bruce Buchanan, Gerald DeJong, Thomas Dietterich, Paul Rosenbloom, and Alex Waibel. Machine learning. *Annual review of computer science*, 4(1):417–433, 1990.

[57] Seyed Amir Naghibi, Hamid Reza Pourghasemi, and Barnali Dixon. Gis-based groundwater potential mapping using boosted regression tree, classification and regression tree, and random forest machine learning models in iran. *Environmental monitoring and assessment*, 188(1):44, 2016.

[58] Hiroki Nakahara, Akira Jinguji, Tomonori Fujii, and Simpei Sato. An acceleration of a random forest classification using altera sdk for opencl. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 289–292. IEEE, 2016.

[59] Hiroki Nakahara, Akira Jinguji, Simpei Sato, and Tsutomu Sasao. A random forest using a multi-valued decision diagram on an fpga. In *Multiple-Valued Logic (ISMVL), 2017 IEEE 47th International Symposium on*, pages 266–271. IEEE, 2017.

[60] Giulio Napolitano, Julia C Stingl, Matthias Schmid, and Roberto Viviani. Predicting CYP2D6 phenotype from resting brain perfusion images by gradient boosting. *Psychiatry Research: Neuroimaging*, 259:16–24, 2017.

[61] Paulino José García Nieto, Esperanza García-Gonzalo, Fernando Sánchez Lasheras, José Ramón Alonso Fernández, Cristina Díaz Muñiz, and Francisco Javier de Cos Juez. Cyanotoxin level prediction in a reservoir using gradient boosted regression trees: a case study. *Environmental Science and Pollution Research*, 25(23):22658–22671, 2018.

[62] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. Demystifying automata processing: Gpus, fpgas or micron's ap? In *Proceedings of the International Conference on Supercomputing*, page 1. ACM, 2017.

[63] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726. IEEE, 2014.

[64] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. Hlspredict: cross platform performance prediction for fpga high-level synthesis. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[65] Muhsen Owaida and Gustavo Alonso. Application partitioning on fpga clusters: Inference over decision tree ensembles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 295–2955. IEEE, 2018.

[66] Muhsen Owaida, Hantian Zhang, Ce Zhang, and Gustavo Alonso. Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2017.

[67] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[68] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[69] Natalia S Podio, María V Baroni, and Daniel A Wunderlin. Relation between polyphenol profile and antioxidant capacity of different argentinean wheat varieties. a boosted regression trees study. *Food Chemistry*, 232:79–88, 2017.

[70] Tao Qin and Tie-Yan Liu. Introducing LETOR 4.0 datasets. *CoRR*, abs/1306.2597, 2013.

[71] J Ross Quinlan. Generating production rules from decision trees. In *ijcai*, volume 87, pages 304–307. Citeseer, 1987.

[72] Stephen Robertson and Hugo Zaragoza. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

[73] Emmanuel Roche and Yves Schabes. *Finite-state language processing*. MIT press, 1997.

[74] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[75] Indranil Roy and Srinivas Aluru. Finding motifs in biological sequences using the micron automata processor. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 415–424. IEEE, 2014.

[76] Indranil Roy, Ankit Srivastava, Matt Grimm, and Srinivas Aluru. Parallel interval stabbing on the automata processor. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pages 10–17. IEEE, 2016.

[77] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. High performance pattern matching using the automata processor. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1123–1132. IEEE, 2016.

[78] Cynthia Rudin and Kiri L Wagstaff. Machine learning for science and society, 2014.

[79] Shruti Sachdeva, Tarunpreet Bhatia, and AK Verma. Gis-based evolutionary optimized gradient boosted decision trees for forest fire susceptibility mapping. *Natural Hazards*, 92(3):1399–1418, 2018.

[80] Ferdinando S Samaria and Andy C Harter. Parameterisation of a stochastic model for human face identification. In *Proceedings of 1994 IEEE Workshop on Applications of Computer Vision*, pages 138–142. IEEE, 1994.

[81] Niek Sanders. Twitter sentiment corpus, 2011.

[82] Christos Sapsanis, George Georgoulas, Anthony Tzes, and Dimitrios Lymberopoulos. Improving emg based classification of basic hand movements using emd. In *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 5754–5757. IEEE, 2013.

[83] Christopher Scarpone, Margaret G Schmidt, Chuck E Bulmer, and Anders Knudby. Semi-automated classification of exposed bedrock cover in british columbia's southern mountains using a random forest approach. *Geomorphology*, 285:214–224, 2017.

[84] Ilya Segalovich. Machine learning in search quality at yandex. *Invited Talk, SIGIR*, 125, 2010.

[85] Robert P Sheridan, Wei Min Wang, Andy Liaw, Junshui Ma, and Eric M Gifford. Extreme gradient boosting as a method for quantitative structure–activity relationships. *Journal of Chemical Information and Modeling*, 56(12):2353–2360, 2016.

[86] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Napel: Near-memory computing application performance prediction via ensemble learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 27. ACM, 2019.

[87] Abdulhamit Subasi, Emina Alickovic, and Jasmin Kevric. Diagnosis of chronic kidney disease by using random forest. In *CMBEBIH 2017*, pages 589–594. Springer, 2017.

[88] Ryutaro Tanno, Kai Arulkumaran, Daniel C Alexander, Antonio Criminisi, and Aditya Nori. Adaptive neural trees. *arXiv preprint arXiv:1807.06699*, 2018.

[89] II Tommy Tracy, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. Nondeterministic finite automata in hardware-the case of the levenshtein automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[90] Da Tong, Yun Rock Qu, and Viktor K Prasanna. Accelerating decision tree based traffic classification on fpga and multicore platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3046–3059, 2017.

[91] Tommy Tracy II. Automatazoo. https://github.com/tjt7a/AutomataZoo, 2018.

[92] Tommy Tracy II, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. Towards machine learning on the automata processor. In *International Conference on High Performance Computing*, pages 200–218. Springer, 2016.

[93] Tommy Tracy II and Mircea Stan. Streaming random forest inference. *GOMACTech*, 2018.

[94] Tommy Tracy II and Mircea Stan. An improved spatial encoding for large decision tree ensemble automata. *GOMACTech*, 2019.

[95] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. *IEEE Transactions on speech and audio processing*, 10(5):293–302, 2002.

[96] Lev Utkin, Andrei Konstantinov, Anna Meldo, Mikhail Ryabinin, and Viacheslav Chukanov. A deep forest improvement by using weighted schemes. In *Proceedings of the 24th Conference of Open Innovations Association FRUCT*, page 63. FRUCT Oy, 2019.

[97] Lev V Utkin and Mikhail A Ryabinin. A siamese deep forest. *arXiv preprint arXiv:1704.08715*, 2017.

[98] Lev V Utkin and Mikhail A Ryabinin. A siamese deep forest. *Knowledge-Based Systems*, 139:13–22, 2018.

[99] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 232–239. IEEE, 2012.

[100] Gill Venner, Stephen Walker, and Nathalie Mitev. Okapi: a prototype online catalogue. *Vine*, 15(2):3–13, 1985.

[101] Cyril Voyant, Fabrice Motte, Gilles Notton, Alexis Fouilloy, Marie-Laure Nivet, and Jean-Laurent Duchaud. Prediction intervals for global solar irradiation forecasting using regression trees methods. *Renewable energy*, 126:332–340, 2018.

[102] Jack Wadden, Kevin Angstadt, and Kevin Skadron. Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 749–761. IEEE, 2018.

[103] Jack Wadden, Nathan Brunelle, Ke Wang, Mohamed El-Hadedy, Gabriel Robins, Mircea Stan, and Kevin Skadron. Generating efficient and high-quality pseudo-random behavior on automata processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 622–629. IEEE, 2016.

[104] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, et al. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–12. IEEE, 2016.

[105] Jack Wadden and Kevin Skadron. Vasim: An open virtual automata simulator for automata processing application and architecture research. *University of Virginia, Tech. Rep. CS2016-03*, 2016.

[106] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, et al. Automatazoo: A modern automata processing benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–24. IEEE, 2018.

[107] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[108] Chenxu Wang, Tingting Cai, Guang Suo, Yutong Lu, and Enqiang Zhou. Distforest: A parallel random forest training framework based on supercomputer. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 196–204. IEEE, 2018.

[109] Ke Wang, Yanjun Qi, J.J. Fox, M.R. Stan, and K. Skadron. Association rule mining with the micron automata processor. In *IPDPS'15*, May 2015.

[110] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Sequential pattern mining with the micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 135–144. ACM, 2016.

[111] Michael HLS Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. Using the automata processor for fast pattern recognition in high energy physics experimentsâĂŤa proof of concept. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 832:219–230, 2016.

[112] Yaozheng Wang, Dawei Feng, Dongsheng Li, Xinyuan Chen, Yunxiang Zhao, and Xin Niu. A mobile recommendation system based on logistic regression and gradient boosting decision trees. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 1896–1902. IEEE, 2016.

[113] Zeyu Wang, Yueren Wang, Ruochen Zeng, Ravi S Srinivasan, and Sherry Ahrentzen. Random forest based hourly building energy prediction. *Energy and Buildings*, 171:11–25, 2018.

[114] Qiang Wu, Christopher JC Burges, Krysta M Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270, 2010.

[115] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. Reapr: Reconfigurable engine for automata processing.

[116] Xilinx. Field programmable gate array (fpga). *URL https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html*.

[117] Xilinx Inc., San José, CA. *Virtual Input/Output v3.0: LogiCORE IP Product Guide*, PG159 edition, 2018.

[118] Weifeng Xu, Jianxin Zhang, Qiang Zhang, and Xiaopeng Wei. Risk prediction of type ii diabetes based on random forest model. In *2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, pages 382–386. IEEE, 2017.

[119] Tommy Tracy II Eric Jonas Yao Fu, Paul Glendenning. Space efficient random forests implementation utilizing automata processors, 2016. Patent No. US20170255878A1, Filed Mar. 7, 2016, PENDING.

[120] Huan Zhang, Si Si, and Cho-Jui Hsieh. Gpu-acceleration for large-scale tree boosting. *arXiv preprint arXiv:1706.08359*, 2017.

[121] Lei Zhao, Quan Deng, Youtao Zhang, and Jun Yang. Rfacc: a 3d reram associative array based random forest accelerator. In *Proceedings of the ACM International Conference on Supercomputing*, pages 473–483. ACM, 2019.

[122] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. Brill tagging on the micron automata processor. In *Semantic Computing (ICSC), 2015 IEEE International Conference on*, pages 236–239. IEEE, 2015.

[123] Zhi-Hua Zhou and Ji Feng. Deep forest: Towards an alternative to deep neural networks. *arXiv preprint arXiv:1702.08835*, 2017.

[124] Xi Zhu, Xiaofei Du, Mike Kerich, Falk W Lohoff, and Reza Momenan. Random forest based classification of alcohol dependence patients and healthy controls using resting state mri. *Neuroscience letters*, 676:27–33, 2018.