RFAcc: A 3D ReRAM Associative Array based Random Forest Accelerator

Lei Zhao
University of Pittsburgh
lez21@pitt.edu

Quan Deng
National University of Defense Technology
dengquan12@nudt.edu.cn

Youtao Zhang
University of Pittsburgh
zhangyt@cs.pitt.edu

Jun Yang
University of Pittsburgh
juy9@pitt.edu

ABSTRACT

Random forest (RF) is a widely adopted machine learning method for solving classification and regression problems. Training a random forest demands a large number of relational comparison and data movement operations, which take long time when using modern CPUs. Accelerating random forest training using either GPUs or FPGAs achieves only modest speedups.

In this paper, we propose RFAcc, a ReRAM based accelerator, to speed up random forest training process. We first devise a 3D ReRAM based relational comparison engine, referred to as 3D-VRComp, to enable parallel in-memory value comparison. We then exploit 3D-VRComp to construct RFAcc to speedup random forest training. Finally, we propose three optimizations, i.e., *unary encoding*, *pipeline design*, and *parallel tree node training*, to fully utilize the accelerator resources for maximized throughput improvement. Our experimental results show that, on average, RFAcc achieves 8564 and 16850 times speedup and 6.6×10^4 and 2.6×10^5 times energy saving over the training on a 4.2GHz Intel Core i7 CPU and a NVIDIA GTX1080 GPU, respectively.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Hardware → Emerging technologies**;

KEYWORDS

ReRAM, Random Forest, Accelerator

ACM Reference Format:
Lei Zhao, Quan Deng, Youtao Zhang, and Jun Yang. 2019. RFAcc: A 3D ReRAM Associative Array based Random Forest Accelerator . In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330387

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330387

1 INTRODUCTION

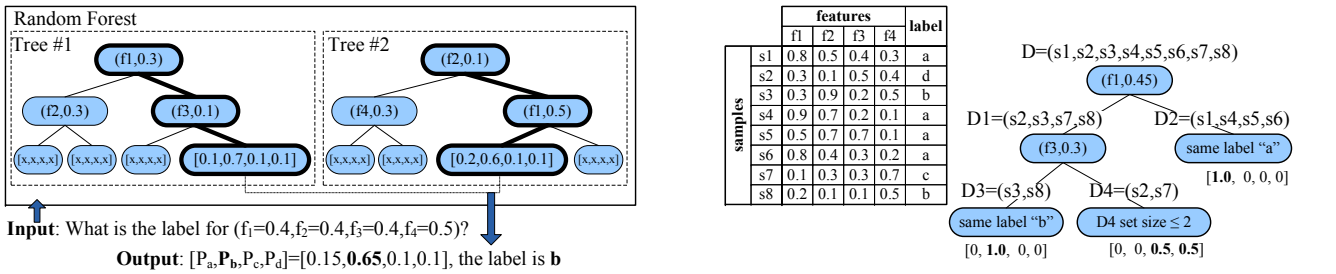
Random forest (RF) [2] is an ensemble machine learning method that makes predictions based on the results of multiple independent decision trees. RF, albeit a simple algorithm, performs very well in solving classification and regression problems, which makes random forest one of the most popular methods in machine learning, data mining and artificial intelligence domains. For example, RF is often the default learning method in Kaggle competitions [15]. As another example, a recent study of 179 classifiers [11] on UCI database [21] and real world problems showed that RF is the best family of classifiers and outperforms other popular classifiers such as SVM [7], neural networks [18] and boost ensembles [24]. RF also has the potential to go deeper. Zhou *et al.* [30] recently built *Deep Forest*, a deep layered RF structure, that outperforms convolutional neural networks on a number of problems that were the home court of the latter.

However, it usually takes a long time to train a RF, e.g., Zhao *et al.* reported that it took up to two days to train on their large data sets using a 2.4GHz Intel Xeon CPU with 16 cores [29]. The training phase is slow because it performs intensive memory accesses as well as relational comparison operations. Training RF on CPUs suffers from limited number of working threads and large branch mis-prediction overhead. Training RF on GPUs suffers from the intrinsic low performance of branch instructions on GPUs. Recent studies showed that GPU-based trainings achieve less than ten times speedup over CPU-based ones [12, 20, 25]. Because of the limited memory bandwidth, accelerating RF training using FPGAs achieves only modest speedup and may have to trade off inference accuracy [4].

Recent studies widely adopt process-in-memory (PIM) to accelerate memory intensive algorithms — PIM avoids massive data movement by performing computation inside the memory. Emerging non-volatile memories, such as ReRAM [1], PCM [3], DWM [27] and STT-RAM [17], have been exploited for PIM acceleration. However, existing designs support either arithmetic operations [6] or match operations [14], which are not suitable for speeding up RF training that is dominated by relational comparisons.

In this paper, we propose RFAcc, a 3D ReRAM based accelerator for RF training. We summarize our contributions as follows.

- We propose 3D-VRComp, a 3D ReRAM based in-memory relational comparison engine. 3D-VRComp compares a set of values saved in the 3D ReRAM arrays with a given input, and splits the value set to those that are bigger than the input and those that are not. To the best of our knowledge,



(a) classification using a random forest

(b) training a random forest with eight samples

Figure 1: The basics of RF. (a) A RF with two decision trees. The input is a four-feature vector $(f1, f2, f3, f4)$ while the prediction output is a label a, b, c or d. (b) Training a decision tree with eight samples. For each sample $(\mathbf{x}_i, y_i)$ (1≤i≤8), $\mathbf{x}_i$ has four features, i.e., f1, f2, f3, f4, while $\mathbf{y}_i$ label can be a, b, c or d.

this is the first ReRAM based relational comparator in the literature.

- We propose to construct RFAcc, a RF training accelerator, by exploiting 3D-VRComp. We further propose three optimizations to minimize data movement and maximize throughput improvement. (1) we adopt unary encoding to improve bit level comparison parallelism; (2) we propose a pipeline design to improve comparison throughput; (3) we concurrently train multiple nodes of a decision tree to fully utilizing the RFAcc hardware resources.
- We compare RFAcc to the state-of-the-art CPU and GPU training implementations. Our experimental results show that, on average, RFAcc achieves 8564 and 16850 times speedup and 6.6×10^4 and 2.6×10^5 times energy saving over the training on a 4.2GHz Intel Core i7 CPU and a NVIDIA GTX1080 GPU, respectively.

In the rest of the paper, we present the RF background and 3D ReRAM basics in Section 2. We present the 3D-VRComp design in Section 3. The full-fledged RF accelerator RFAcc is described in Section 4. We elaborate our three level parallelisms in Section 5. Section 6 and Section 7 discuss the experiment methodology and the results, respectively. Finally we conclude the paper in Section 8.

2 PRELIMINARIES

2.1 Random Forest

2.1.1 Random Forest Classification. Random forest (RF) is an ensemble machine learning method for classification, regression and many other tasks. A random forest consists of multiple decision trees while each tree is a weak learner. Each decision tree, with its accuracy being barely above chance, requires only simple computation. By aggregating a number of decision trees into a forest and averaging the outputs of all the trees as the final output, the overall accuracy can be greatly improved.

Figure 1(a) illustrates a RF with two decision trees. It takes inputs with numeric values for four features $(f1, f2, f3, f4)$ and predicts the output label being a, b, c or d. Each internal tree node (including the root node) is marked as (f_i, v_i) indicating how to walk the tree with a given input. For example, with input $(f1, f2, f3, f4) = (0.4, 0.4, 0.4, 0.5)$ and the root node of the first tree $(f_1, 0.3)$, we walk down the right subtree as the input's f_1 feature value is bigger than 0.3 (otherwise, we walk down the left subtree). The walking continues until it reaches the leaf node. The latter is marked with a probability vector (P_a, P_b, P_c, P_d) for the four labels a, b, c, or d. In the figure, the first tree predicts that the input has 0.1, 0.7, 0.1 and 0.1 probabilities of being label a, b, c or d, respectively.

Given that we have two trees, we average the prediction probabilities from two leaf nodes and pick up the label with largest probability. For the example, the final prediction probability vector is (0.15, 0.65, 0.1, 0.1) such that the RF outputs the predicted label being b.

2.1.2 Decision Tree Training. We next briefly discuss how to train a decision tree and then build a RF.

To train a decision tree, we prepare a training set with n samples $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)\}$. Each sample $(\mathbf{x}_i, y_i)$ is composed of m features $\mathbf{x}_i = \{x_i^{(1)}, x_i^{(2)}, ..., x_i^{(m)}\}$ and one label $y_i \in \{1, 2, ..., K\}$ (i.e., K classes). The features take numeric values while the label is from a label set. In Figure 1(b), we have eight samples and each $\mathbf{x}_i$ has four features and the label can be a, b, c or d.

Training a decision tree is to incrementally create tree nodes and pick up the feature-value pair for each internal node of the tree. When training a tree node, we try a subset of features with different values for each feature and then pick up the best one from these tries. For example, for the root node, we may choose $(f_1, 0.45)$ such that the training set D is split to two subsets D_1 and D_2 as follows.

$$D_1 = \{(\mathbf{x}_i, y_i) | x_i^{(f_1)} \leq 0.45\}$$
$$D_2 = \{(\mathbf{x}_i, y_i) | x_i^{(f_1)} > 0.45\} \quad (1)$$

After fixing the feature-value pair for an upper level tree node, we continue training its subtree nodes, and stop if a subtree node's sample set has fewer than a threshold samples, or all the samples in the tree node's sample set have the same label. We construct the probability vector based on the percentage of samples having each label.

For the example in the figure, after training the root node, D_1 has four samples with different labels so that we may continue training this subtree. However, D_2 contains four samples and all samples have the label a. We therefore stop training the right subtree. The predict probability vector is computed as (1.0, 0, 0, 0) as all samples have a. For the leaf node represented by subset D_4, assume we stop here as the set size is below a threshold. Its predict probability

vector is computed as (0, 0, 0.5, 0.5) because half of the samples have label c while the other half having d.

Fixing the feature-value pair. To determine the appropriate feature-value pair for an internal tree node, we try many features and try many values for each feature. We then pick up the one having the highest homogeneity. CART is a commonly used algorithm that uses *Gini Impurity* to determine the feature-value pair.

Given one feature-value pair, a node splits the sample set D that reaches this node into subsets D_1 and D_2. CART tries all feature-value pairs (i.e. all features f_i ($i = 1, 2, ..., m$) and all possible values v_j for that feature) to split D and then computes the *Gini Impurity* as follows.

$$Gini(D, f_i, v_j) = \frac{|D_1|}{|D|}(1 - \sum_{k=1}^{K}(\frac{|D_{1k}|}{|D_1|})^2) + \frac{|D_2|}{|D|}(1 - \sum_{k=1}^{K}(\frac{|D_{2k}|}{|D_2|})^2) \quad (2)$$

Where $|D|$, $|D_1|$ and $|D_2|$ are the sizes of the corresponding subsets; $|D_{1k}|$ and $|D_{2k}|$ are the number of samples in D_1 and D_2 with label $y = k$, respectively. CART selects the feature-value pair (f_i, v_j) that minimizes Equation 2 as the splitting rule for the node. By rewriting Equation 2, it is equivalent to choosing the (f_i, v_j) pair that maximizes the following.

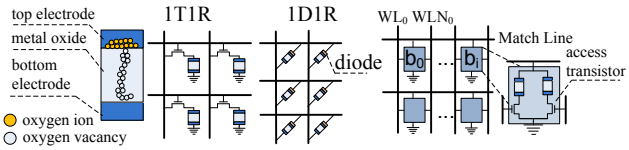
$$\frac{1}{|D_1|}\sum_{k=1}^{K}(|D_{1k}|)^2 + \frac{1}{|D_2|}\sum_{k=1}^{K}(|D_{2k}|)^2 \quad (3)$$

From above discussion, trying one feature-value pair consists of (i) splitting the sample set and (ii) computing the *Gini Impurity*. The former demands O(|D|) comparisons while the latter demands counting the sizes of K subsets and O(K) add / multiplication operations. Our design accelerates both comparison and set counting with in-memory operations while using integrated ALU units to accomplish the add/multiplication computation.

Ensemble of decision trees. Although a single decision tree works poorly, studies showed that the overall accuracy can be greatly improved if we aggregate a number of decision trees into a forest and average the prediction probability vectors of all the trees as the final output. RF exploits this observation and constructs uncorrelated decision trees with two adjustments in training: (i) when training a decision tree, it uses a randomly chosen subset of samples rather than all samples; (ii) when training an internal tree node, it uses a randomly chosen subset of features rather than all features.

2.2 ReRAM and 3D ReRAM based TCAM

2.2.1 ReRAM and TCAM. ReRAM (Resistive Random Access Memory) is an emerging non-volatile memory technology. A ReRAM cell is made of metal oxide material that is sandwiched between the top and bottom electrodes, as shown in Figure 2(a). With different injected currents, the cell may have oxygen vacancy filament constructed or destructed in the oxide material. The cell exhibits low resistance R_L and high resistances R_H when having and not having the filament, representing logic '1' and '0', respectively. Recent studies have architected ReRAM cell arrays as either storage or computing unit. ReRAM based storage often adopt 1T1R or 1D1R cell structures, as shown in Figure 2(b).



(a) ReRAM device (b)ReRAM cell arrays (c) ReRAM 2D TCAM

Figure 2: The ReRAM basics and 2D TCAMs.

Two ReRAM based computing units are studied in the literature. One is to exploit the natural current accumulation in ReRAM arrays to speedup dot-product computation [6]. The other is to construct Ternary Content-Addressable Memory (TCAM) compute engines [13]. We next briefly discuss ReRAM based TCAM design.

Figure 2(c) illustrates a TCAM array with each row saving n-bit data. We program a pair of cells to complementary states to represent each saved bit, i.e., the two cells are programmed to either (R_H, R_L) or (R_L, R_H) to represent logic '1' or '0', respectively. Given an n-bit input, we can compare it to all rows simultaneously. For the i-th input bit (0≤i≤n-1), we set $\mathtt{WL}_i$ and $\mathtt{WLN}_i$ to the input and its complementary, respectively. We use high and low voltages to represent logic '1' and '0', respectively.

A match line is precharged to high voltage before comparison and exhibits voltage drop only if at least one of the TCAM cells along the corresponding row *mismatches* the input bit. That is, since $\mathtt{WL}_i$ and $\mathtt{WLN}_i$ opens one transistor for each cell pair, a *mismatch* occurs if the ReRAM connected to the opened transistor is in R_L state, which discharges the current and brings down the voltage of the match line. The match line remains at the high voltage if the data saved in the corresponding row matches the input bit-by-bit. So, the conventional TCAM can only check whether two data are equal or not, while not able to distinguish which data is bigger or smaller (i.e., the relational comparison).

2.2.2 3D ReRAM based TCAM. Recent advances in ReRAM proposed 3D ReRAM structures, i.e., building ReRAM arrays along the third dimension, to further increase bit density [5]. There are two approaches. One is to stack planar cross-point structure layer by layer while the other is to construct 3D Vertical ReRAM (3D-VRRAM) structure. Since the former does not scale well and the latter has low per-bit cost[9, 28], we adopt 3D-VRRAM in this paper.

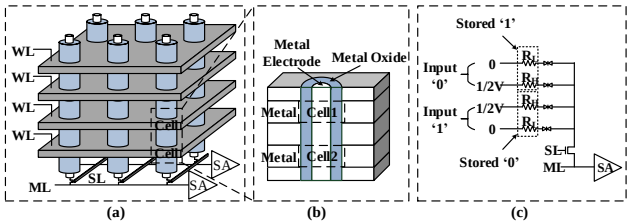


Figure 3: 3D ReRAM based TCAM [19].

Figure 3(a) illustrates a 4-layer 3D-VRRAM architected for TCAM operation [19]. The four cells in one column share one metal electrode (see Figure 3(b)) such that they can be enabled when the access transistor at the bottom (controlled by the sourceline) is enabled. For the data saved in the TCAM, each saved bit is represented using two cells (in complementary states) from two adjacent layers in the same column, as shown in Figure 3(c), the upper two cells

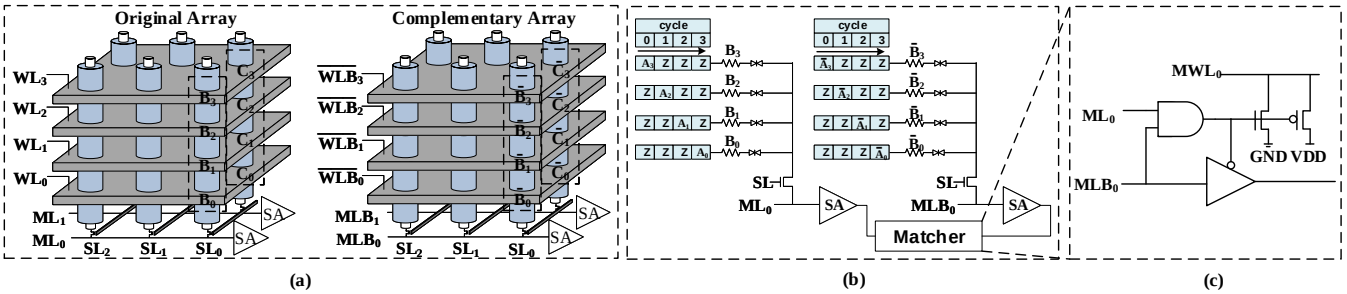


Figure 4: The 3D-VRComp relational comparator.

store a '1' and the lower two cells store a '0'. When we encode the input using the wordlines (i.e., each input bit and its complementary connect to two wordlines) and enable one sourceline, we compare the cells from one vertical plane with the input. Each matchline at the bottom indicates if the input matches the saved data in one column.

3 3D-VRCOMP: 3D RERAM BASED RELATIONAL COMPARATOR

While both 2D and 3D ReRAM based TCAM designs allow parallel in-memory matching, they do not support relational comparisons that we need in RF training. This motivates our design of a novel in-memory relational comparator, i.e., the greater- or smaller-than relationships can be quickly determined. In this section, we devise a 3D-VRRAM based relational comparator engine, referred to as 3D-VRComp, as the critical building block in RFAcc.

To simplify the hardware design, we preprocess the feature values in the training set as follows. (1) We convert all values to non-negative values, i.e., a feature's value range is changed from $[-a, +b]$ to $[0, a+b]$ with simple adjustment. (2) We represent the values in 32-bit fixed point numbers. This is sufficient for the benchmarks that we tested. If a feature's value range is too big, we may adopt value normalization to represent the values in [0,1]. Given the feature-value pair applied at an internal tree node is to split the sample set, applying above two value transformations shall not alter the training difficult or the final result. It is clear that we also need to apply the same value transformations to the inputs before the real task.

The basic strategy employed in 3D-VRComp is to compare bit-by-bit. That is, when comparing two n-bit values $A_{n-1}...A_1A_0$ and $B_{n-1}...B_1B_0$ (A_{n-1} and B_{n-1} are the most significant bits), we start from comparing A_{n-1} and B_{n-1} and proceed to compare the next bit only if A_{n-1}=B_{n-1}; otherwise, the final result takes the comparison result of A_{n-1} and B_{n-1}. If the comparison stops at A_0=B_0, we have A=B.

Figure 4 presents the structure of 3D-VRComp. Assume we are to compare A with a set that contains B and C. All values have four bits, e.g., A=$A_3A_2A_1A_0$. We save B and C in the cell array (only use one vertical plane as in Figure 4(a)), have A as the input, and output the matched items in the set. For each saved bit of B and C, 3D-VRComp saves the original bit and its complementary in two separate arrays (denoted as, e.g., C_i and $\bar{C_i}$ in 4(a)).

Figure 4(b) shows the equivalent circuit for comparing A with B. To compare the first bit, i.e., comparing A_3 to B_3, we activate the sourceline for B, i.e., SL_2=SL_1=0 and SL_0=1 in the example; we charge all matchlines to high voltage, i.e., ML_0= MLB_0=1; In the first cylce, we place the to-be-compared bit and its complementary bit on one wordline, i.e., WL_3=A_3, WLB_3=$\bar{A_3}$, and WL_2 = WLB_2 = WL_1 = WLB_1 = WL_0 = WLB_0 = Z (Z indicates disconnected input). We then use R_H and R_L to represent logic '0' and '1' in ReRAM cells; and use $0V$ and $\frac{1}{2}V$ to represent logic '0' and '1' of the input, respectively.

Given that one comparison generates two matching results, e.g., ML_0 and MLB_0 hold the results of A_3 and B_3 comparison, we can differentiate all three possibilities, i.e., A=B, A<B, or A>B. This is impossible in the traditional one-matchline TCAM design — one matchline can only differentiate two states (i.e., equal or not equal). For both arrays, if wordline is 0V, matchline voltage drops only if the cell has R_L resistance. Therefore, we have A_3=B_3 if both matchlines hold high voltages after comparison; A_3>B_3 if ML_0 holds the high voltage while MLB_0 drops to the low voltage; and A_3<B_3 if MLB_0 holds the high voltage while ML_0 drops to the low voltage. In the subsequent cycles, we compare A_2 to B_2, A_1 to B_1 and A_0 to B_0 one by one if the previous results are all equal. Because C use different SA and matcher, the comparison between A and C can take place simultaneously with comparison between A and B. Figure 4(c) shows the circuit of the matcher.

4 THE RFACC DESIGN

4.1 An Overview

In this section, we exploit 3D-VRComp to construct a full-fledged RF accelerator (RFAcc) to speedup RF training. We first present an overview and then elaborate each building component.

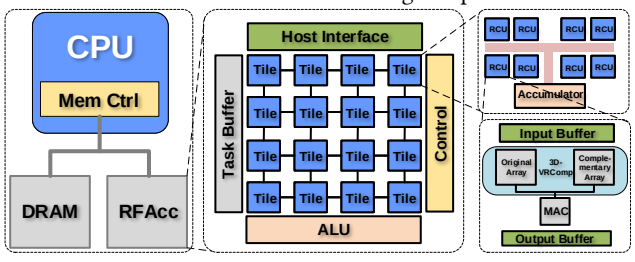


Figure 5: The proposed RFAcc architecture.

To train a RF, the host CPU sends a configuration file to RFAcc such that RFAcc trains the whole forest asynchronously and sends the trained forest back to the host. The configuration file is written by the programmer, which defines the parameters of the target forest (e.g., the number of trees and the maximum depth of each tree, etc.) and the characteristics of training data (e.g., the number

of samples and the number of features, etc.). We assume the training input has at most 2^{20} samples[1].

Figure 5 elaborates the RFAcc architecture. RFAcc is integrated into the system as a memory module. A RFAcc chip is composed of a Task Buffer, an ALU, the control logic and an array of tiles. RFAcc trains one decision tree at a time but may train multiple nodes of this tree simultaneously with our later optimization. The Task Buffer records the information of tree nodes being and to be trained. The ALU is to compute *Gini Impurity* to determine if a feature-value pair is a good choice for a split. Each tile is composed of a number of RCUs (Relational Compare Units) to compare the input with randomly selected features, and an accumulator to accumulate the partial results from RCUs. The control logic orchestrates the work in different components.

4.2 The Building Blocks

4.2.1 RCU. A RCU (Relational Compare Unit) has a 3D-VRComp to store the samples. Each 3D-VRComp adopts two 64×128×128 3D ReRAM arrays (for the original and complementary data, respectively), i.e., it has 64 layers, 128 matchlines per array, and 128 source lines. One 3D-VRComp can save 128 samples, 256 features of each sample, and 32 bits per feature. For example, we assume that a training set has 512 samples and each sample has 512 features. We use 8 RCUs to save the samples, as shown in Figure 6. Sample 0 to 127 expand across RCU_0 and RCU_1, sample 128 to 255 expand across RCU_2 to RCU_3, etc. RCU_{2i} store feature 0 to 255 and RCU_{2i+1} store feature 256 to 511, where $0 \le i \le 3$.

A RCU has an Input Buffer storing three bit vectors `member_bv`, `best_gini_bv`, and `current_split_bv`. While one RCU saves consecutive 128 samples, not all of them belong to the node being trained. `member_bv` is the member bit vector denoting the samples that are the ones in node's sample set. Given a feature-value pair, the 3D-VRComp splits the sample set into left subtree and right subtree. `current_split_mask` records the member bit vector of the left subtree for the feature-value pair being tried. The right subtree's member bit vector can be calculated by

$$\neg \texttt{current\_split\_mask} \wedge \texttt{member\_bv}$$

. `best_gini_mask` records the member bit vector of the left subtree that calculates the best *Gini Impurity*.

The RCU also has a MAC array to count the labels in each sample subset. We have discussed how the 3D-VRComp engine works in Section 3. We next elaborate MAC details.

4.2.2 MAC unit. For the 128 samples saved in one RCU, the MAC is a 2D ReRAM crossbar storing their corresponding labels. The labels used in the training set are encoded as 1-hot vector values. Each label is a 64-bit vector that has a unique element being 1 and all others being 0s. For example, labels *a*, *b*, *c* and *d* are encoded as (1,0,0,0,...), (0,1,0,0,...), (0,0,1,0,...) and (0,0,0,1,...), respectively. In this paper, we set K=64[2], so the MAC crossbar array size is 128×64. We feed the wordlines with the subtree's member bit vector produced by the 3D-VRComp as input. The accumulated current on bitlines are the output indicating the label count. For the example in Figure 7, after a split, the subtree has 12 samples with label *a*, 5 samples with label *b*, 9 samples with label *c*, etc. We use 32 ADC units to finish the counting in 2 rounds.

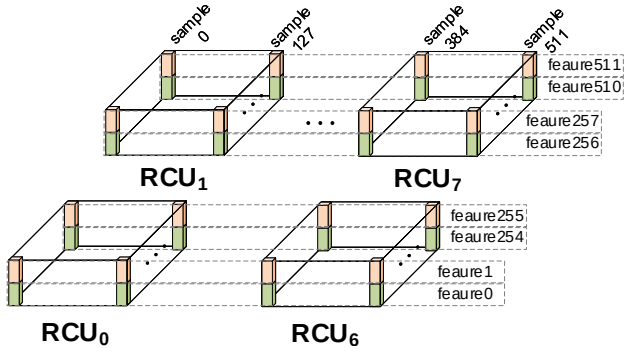


Figure 6: Data mapping in RCUs.

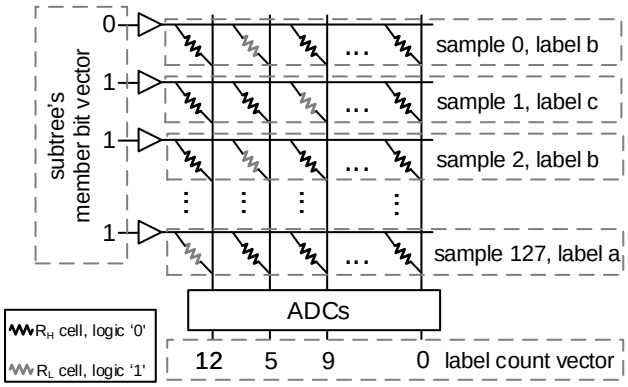


Figure 7: MAC.

4.2.3 Tile. Multiple RCUs are grouped in a tile. To ease the burden of ALU and NOC overhead, after the RCUs splits a node, the label count are accumulated in the accumulator. So, only one accumulated label count is sent from each tile to the global ALU through the 2D mesh NOC.

Table 1: Task Buffer Fields

Field Name	Size	Description
node_ID	4B	ID of node starting from 1
input_group_bv	4KB	Each bit indicates a group of 128 samples
mask_bv	512KB	Sample subset of this node (one bit per sample)
RCU_request_bv	4KB	Requested RCUs (one bit per RCU)
feature_seed	4B	Seed for randomly selected features
best_feature_value	10B	Feature (16b) and value(32b) that achieves the best Gini(32b)
working_feature_value	6B	Currently tried feature (16b) and value(32b)

4.2.4 Task buffer. A task buffer is a 64-entry SRAM buffer with each entry containing the fields shown in Table 1.

The first three fields describe the node characteristics. The ID of the root node is set to '1'. Since a decision tree is a binary tree, the

[1]We assume one RFAcc can load all samples and their features. For large training sets, we priority loading samples so that we may load only a subset of features per sample. We leave it as our future work to develop dynamic swapping schemes to address this issue.

[2]This is sufficient for our training set. We need more counting rounds if there are more label classes.

IDs of the two subtree nodes of an internal node with ID x are set to $2x$ and $2x+1$, respectively. We assign each sample in the training input set with its appearing order number (starting from 0). Since a node during training contains only a subset of all samples, we use two-level bit vectors to denote its members. For every 128 samples, we use one bit in `input_group_bv` to indicate if any of these 128 samples appears in the node's sample set. For each non-zero bit j, `mask_bv` saves a 128-bit bit vector at offset j×128 to identify which samples in this group are in the node's sample set. Clearly, `mask_bv` reserves storage for the worst case while we may use only a small portion at runtime.

The next two fields describes the training feature selection. Since training a tree node needs to randomly try $\sqrt{F}$ features (F is the total number of all features), we record the random seed to generate these features in `feature_seed`. Since each RCU can hold 256 features of a sample, we need more RCUs to save the features from each sample if there are more than 256 features. The `RCU_request_bv` records which RCUs may be used to train the node. Still take the 8 RCUs in Figure 6 as an example. Training one node needs to use $\sqrt{F} = 22$ features. If all the 22 features are from the first half, `RCU_request_bv`='01010101'; if all from the second half, `RCU_request_bv`='10101010'; otherwise, `RCU_request_bv`='11111111'.

The last two fields describes the training progress. `working_feature_value` saves the current feature-value pair being tried. `best_feature_value` saves the best *Gini Impurity* value and its corresponding feature-value pair during training.

4.3 Training A Random Forest

We next use an example to elaborate how RFAcc trains a RF (Figure 8). Assuming we have a training set with 1280 samples and each sample has 1024 features. We first load these samples to RFAcc with each sample expands across four RCUs, that is, samples 0 to 127 occupy RCU_0 to RCU_3 while samples 128 to 255 occupy RCU_4 to RCU_7, etc. RCU_0 and RCU_4 saves the first 256 features of their corresponding samples. 40 RCUs in one tile are enough to hold all the samples.

(1) Task buffer entry initialization. To train a decision tree, we generate a root node in the task buffer with `node_ID`=1, `input_group_bv` =0x0...03FF (i.e., the root node contains all the 10 sample groups), and `mask_bv` being 0xFF...FF (1280 1s). We continuously process the entries in the task buffer until the buffer is empty. We reserve two empty entries before processing one entry.

Training a node needs to try $\sqrt{1024}$=32 random features. Assume the seed for the random generated features is 13, and these features are features 0 to 15, and 256 to 271, we set `RCU_request_bv`=0x3333333333 indicating we use RCU_{4i} and RCU_{4i+1} ($0 \le i \le 9$).

(2) Select RCUs. According to the `RCU_request_bv` field of this node in task buffer, RCU_{4i} and RCU_{4i+1} (marked by shade in Figure 8) are selected to perform the task.

(3) Initialize RCUs. The involved RCUs initialize their `member_bv` registers according to `input_group_bv` and `mask_bv` in task buffer, load member bit vectors from `mask_bv` with offset 128×i into their `member_bv` registers. For example, because RCU_0 stores the first 128 samples, it checks the first bit in `input_group_bv`, if the first bit is 1, then load the first 128 continuous bits from `mask_bv` to its `member_bv`, otherwise initialize its `member_bv` to 0. Since it is the root node which needs to split all the samples, the `member_bv` is initialized to 0xF...FF (128 1s).

(4) Comparison. We then try all the chosen features and try all value choices for each feature using the relational comparison capability of 3D-VRComp. For each split, a 128-bit `current_split_mask` is generated, indicating which samples are split into the left child node. In the example, we assume the `current_split_mask` is 0xA...57.

(5) Counting labels in each subtree. We next generate the subtree nodes' member bit vectors. '`member_bv` ∧ `current_split_bv`' produces the `member_bv` for its left subtree; and '`member_bv` ∧ ¬`current_split_bv`' produces the `member_bv` for its right subtree. We first send the left subtree's member bit vector to the MAC unit. The latter exploits the current-accumulation characteristic of ReRAM [26] to measure the current of each bitline. The result indicates the number of corresponding labels in left subtree node's sample set. The example in Figure 8 shows the left subtree has 12 samples with label a, 5 samples with label b, etc. We repeat this process by using the right subtree's member bit vector to get the right subtree's label count.

(6) Computing Gini. The two label count vectors are then sent to the global ALU to compute the *Gini Impurity* according to Equation 3.

(7) Initializing the subtree nodes. If it is better than the best of previous tries. We record the *Gini Impurity* and the feature-pair in the task buffer. In each involved RCU, we overwrite the `best_gini_bv` with `current_split_bv`. After training one node, we update the two subtree nodes in the reserved task buffer entries. The `node_IDs` are 2 and 3. We then update the `mask_bv` for the 1s in the current node's `input_group_bv` and copy `best_gini_bv`. For one subtree node, if the RCU's `best_gini_bv` (or its complementary AND `mask_bv`) are all 0s, we clear the corresponding bit in subtree node's `input_group_bv`.

We then send the trained feature-value pair for node 1 back to the host CPU and clear the entry in the task buffer, which concludes the training of one tree node.

Since we need to reserve two entries in the task buffer before splitting a node, and after the splitting only one entry is released. It is possible that the task buffer is exhausted. In such case, we offload the whole task buffer to host memory, leaving only the deepest node in task buffer. The following training process only splits the subtree starting from this node.

When preparing the subtree nodes in the task buffer, we skip filling the node if it has too few members, or all labels are the same.

5 OPTIMIZATIONS

5.1 Bit Encoding

RFAcc speeds up RF training by enabling multiple sample comparisons simultaneously. On the one hand, one sample comparison is still slow as it is done bit-by-bit; on the other hand, many RCUs are idle if the number of samples and the number of features per sample are not big.

Figure 9 illustrates why bit-by-bit comparison is necessary for comparing two binary values. In particular, comparing '0011' and '0101' generate conflicting results at two bit positions, discharging both `ML` and `MLB`. The red arrows in the figure show the paths that

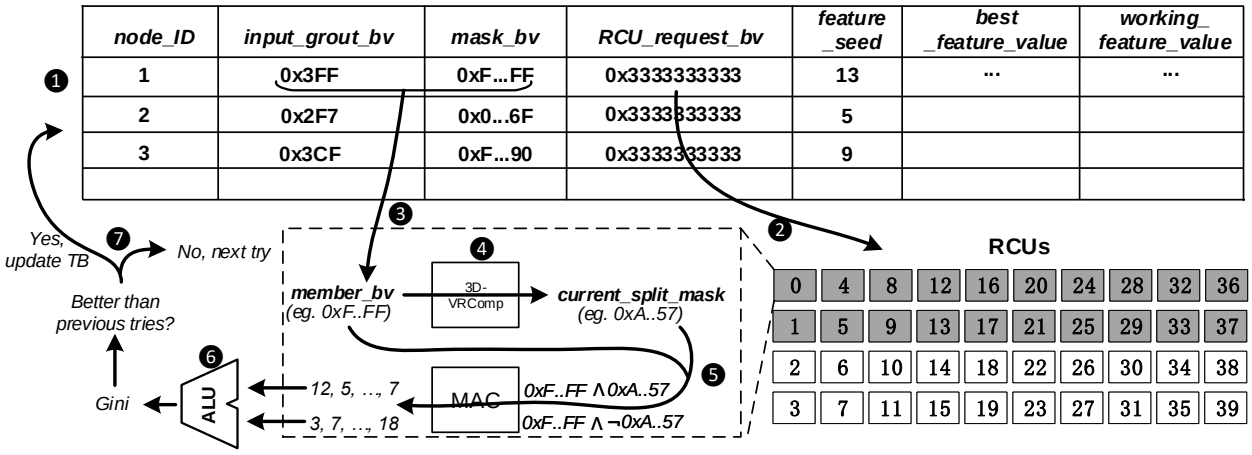


Figure 8: A training example.

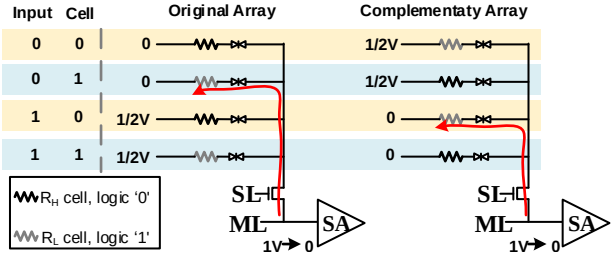


Figure 9: The basic RFAcc demands sequential comparison.

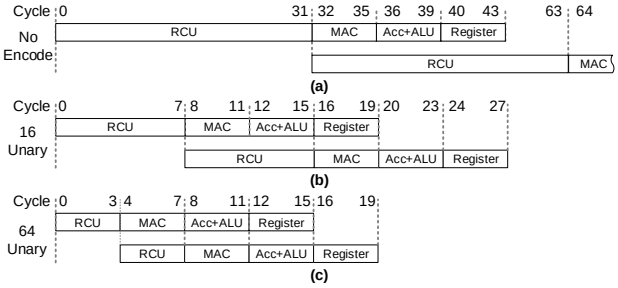


Figure 10: Pipeline.

discharge current on ML and MLB. This comparison result indicates unknown result. Therefore, parallel comparison is not supported in the basic RFAcc.

In this section, we adopt *bit encoding* to improve comparison parallelism — we encode feature values using unary codes so that we can compare multiple bits from one sample simultaneously. A 4-bit generalized unary code [16] represents every two consecutive bits in the original value — bit combinations 00/01/10/11 are converted to 0000/0001/0011/ 0111, respectively. For example, the 4-unary code for binary input '0110' is '0001 0011'.

Adopting unary code enables parallel comparison as the comparison of non-equal bit positions are always consistent. For example, when comparing '0001' and '0111', we have equal comparison results for the first and the fourth bit positions, and the same '0<1' non-equal result for the second and the third bit positions. Given equal comparison does not discharge matchline, we can get the consistent comparison result if the two values are not the same.

Adopting unary encoding reduces area efficiency as we need to use 64 bits to encode the original 32 bit value. However, it improves comparison performance as we finish the comparison of two bits in one comparison step. In general, for 32-bit value comparison that finishes in 32 steps, adopting 2^M-unary code demands $2^M \times \frac{32}{M}$ bits and finish the comparison in $\frac{32}{M}$ steps.

5.2 Pipeline

Training a tree node needs to try a large number of feature-value combinations such that it often takes a long time to finish. A careful study of each try reveals that it includes the following steps. (1) given a feature-value pair, the 3D-VRComp splits the sample set into two subsets, producing the left subtree's member bit vector; (2) the left subtree's member bit vector are used as input to drive the MAC to count left subtree's labels, then similarly, the MAC count the right subtree's labels; (3) the label counts produced by all the RCUs are accumulated in the tile and sent to the global ALU to compute the *Gini Impurity*; (4) if the *Gini Impurity* is better than the best of all previous tries, saving the current split result in each involved RCU. Given that these four steps use different physical functional units, we pipeline their execution for maximized throughput. If a feature-value pair does not give a better split, stage (4) could be skipped but we still keep its cycles in the pipeline to simplify the control overhead.

The cycle time is determined by the slowest stage, i.e. stage (1) which involves current sharing through ReRAM cells. We set the cycle time to 12ns according to [19]. The length of stage (1) is determined by the encoding scheme. As shown in Figure 10(a), if no encoding is used, stage (1) requires 32 cycles to compare the bits in serial. The remaining stages can be hidden in the next comparison. So after set-up phase of the pipeline, each try needs 32 cycles. If a 16-unary encoding is used, the comparison only needs 8 cycles, however, the bottleneck is still stage (1), as shown in Figure 10(b). More aggressively, if there is enough space we can use 64-unary encoding, the length of stage (1) is the same as other stages, every unit can keep busy to produce the highest throughput, as shown

in Figure 10(c). To support encoding in pipeline, the configuration need to be determined offline and loaded into the control logic to drive the finite state machine.

5.3 Node Level Parallelism

Section 4.2 presents the sequential training, that is, the whole RFAcc trains one tree node even if it only uses a subset of all RCUs. To further improve training performance, we propose to enable node level parallel training.

We use a global bit vector `free_RCU_bv` to track free RCUs at runtime. Training a tree node needs to reserve all its needed RCUs. We derive the requested RCUs from `input_group_bv` and `RCU_request_bv`, reserve these RCUs if they are idle, and then start training. Another node may start training only if it can reserve all its needed RCUs; otherwise, it has to wait. The node level parallelism tends to be limited at the beginning and increases as we train the nodes towards the leaves. Training a node close to the leave require few RCUs as the node's sample set tends to be small. We set to train at most 16 nodes at the same time. Since the accumulator is shared by all the RCUs in a tile, and the global ALU is shared by all the tiles, we increase the number of accumulators and ALUs accordingly to support the parallel training.

In this paper, we schedule the training of the nodes recorded in the task buffer sequentially and pause the parallel training if the next node cannot reserve all its requested RCUs. A more aggressive approach is to dynamically search the ready nodes in the task buffers and train out of the order. We will evaluate its complexity and performance tradeoff in our future work.

6 METHODOLOGY

We evaluated the effectiveness of our proposed RFAcc accelerator by comparing it with publicly available random forest training implementations on both CPU and GPU. For CPU implementation, we used RandomForestClassifier from scikit-learn [23] on an Intel Core i7-7700K processor. For GPU implementation, we used CudaTree [20] on a GTX1080 GPU. We used RAPL [8] and Nvidia-SMI [22] to measure CPU and GPU power consumption, respectively.

To model RFAcc, we first used scikit-learn to generate the traces of the trained RF, then we feed the traces into our cycle-accurate RFAcc simulator to get the performance and energy statistics. We used Design Compiler with 32nm technology node to generate latency and power parameters and estimate the area for logic units. The parameters for SRAM buffers and 3D-VRRAM arrays are generated using NVSIM [10]. The specification details are listed in Table 2. We set 2 as the minimum number of samples to stop node split. There is no limitation for the depth of the trees in RF.

Benchmarks. We tested ten benchmarks from publicly available datasets, their characteristics are list in Table 3. Most of the datasets are available in UCI [21] database, which has been widely used by researchers in machine learning community. In addition, we also used two image datasets to test RFAcc with large number of features — mnist is a hand-written digit dataset; orl contains face images of 40 persons, each face is a 92×112 gray scale image. For the image datasets, each raw pixel is treated as a feature in RF. The benchmarks also have large number of samples. For instance, poker covtype have 100M and 58M samples, respectively.

Table 2: Hardware Specification

CPU (Core i7-7700K)	Base Frequency	4.20 GHz
	Cores/Threads	4/8
	Process	14 nm
	TDP	91 W
	Cache	8 MB SmartCache
	System Memory	16 GB DRAM, DDR4
GPU (GTX1080)	Frequency	1733 MHz
	Cuda Cores	2560
	Process	16 nm
	TDP	180 W
	Cache	2MB shared L2
	Graphic Memory	8 GB DRAM, GDDR5X
RFAcc	Task Buffer:32MB, RCU:8GB, MAC:32MB, R_H:10MΩ, R_L:100KΩ, t_{Read}:11.2ns, t_{Write}:25.2ns, V_{Read}:0.4V, V_{Write}:2V	

Table 3: Benchmarks

Benchmark	# of samples	# of Feature	# of Classes
poker	1000000	10	10
covtype	581012	54	7
adult	32561	14	2
iris	150	4	3
letter	20000	16	26
pendigits	7494	16	10
yeast	1484	8	10
mnist	60000	784	10
orl	400	10304	40
intrusion	125973	41	23

Schemes. We compared the following schemes with CPU and GPU based training baselines.

- `RFAcc`. This is our basic RFAcc implementation as elaborated in Section 4 with no encoding and node parallel optimizations, the pipeline execution is enabled by default.
- `RFAcc-X`. This is the implementation after adopting X-unary encoding optimization, i.e., encoding $log_2(X)$ binary bits to X bits (X can be 4, 8, 16, 32 or 64).
- `RFAcc-P`. This is the implementation that enables multiple node training in one RFAcc chip.
- `RFAcc-X-P`. This is the implementation with all optimizations, i.e., X-unary encoding and multiple node training.

7 EVALUATION

7.1 RFAcc Characteristics

Table 4 lists the area and power consumption of a RFAcc chip using 64×128×128 3D ReRAM arrays. One chip can accommodate 32768 RCUs, which occupies 98% chip area. The overall chip area and power consumption are comparable to a ReRAM based accelerator for speeding up CNNs [26]. We use 32nm technology node, the total chip area is $75mm^2$ with 30W power consumption.

Table 4: RFAcc Characteristics

Units	Number/Size	Area (mm^2)	Power
RCU (32768 RCU on chip)			
RComp Units	64×128×128	0.0015	0.89mW
MAC	128×64	0.0012	500uW
I/O buffers	1	1.15e-5	1.5nW
RCU Total	1	0.0027	0.9mW
RCUs	32768	86.8	29.3W
CTRL	1	0.14	31mW
Task buffer	1	0.128	61.2mW
ALU	1	0.599	161.492mW
Chip Total	1	87.75	30W

7.2 Performance

Figure 11 compares the speedup of different schemes. The results were normalized to the CPU baseline. The Y-axis is drawn in log scale. The GPU implementation can only outperform CPU for benchmarks which have more than millions of samples, e.g., poker, covtype, mnist and intrusion,. What's more, GPU can only achieve less than ten times speedups. For small datasets, the GPU implementation has less parallelizable potentials and thus becomes worse than CPU implementation.

For all benchmarks, RFAcc based schemes achieve significant speedup over CPU and GPU baselines. `RFAcc`, which does not have encoding and node parallel optimizations, can achieve 482× speedup on average. When node parallelism is enabled, the average speedup boosts to 1615× (`RFAcc-P` in Figure 11). Because iris has a very small number of samples and features, there is little opportunity for RFAcc to exploit the node parallelism. For orl, although it has more than 10k features, the small number of samples limits the parallelism (the 400 samples expands only 3 RCUs). Beacuse 64-unary encoding reduces the comparison round 8 times, which is the most computational intensive step in RFAcc, `RFAcc-64` improves the performance on all benchmarks. On average `RFAcc-64` has a speedup of 2558× over CPU baseline. When all optimizations are enabled, `RFAcc-64-P` boosts the speedup to 8564×.

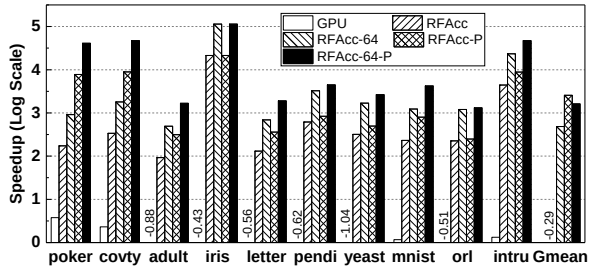


Figure 11: Speedup normalized to CPU.

7.3 Energy Savings

We then evaluated the energy savings in RFAcc. Figure 12 summarizes the energy savings over the GPU baseline. From the figure, training using GPU consumes more energy than that using CPU — it consumes about 2× energy on average. The smaller energy consumption on poker and covtype is because of the shorter execution time on GPU.

From Figure 12, RFAcc shows its superior energy-efficiency over GPU and CPU. The energy advantage of RFAcc comes from its PIM characteristic which avoids massive data movement, and the vast parallelism of feature comparison which is the most time and energy consuming operation in RF training. `RFAcc` and `RFAcc-P` achieve 10^5 energy savings on average. With encoding, `RFAcc-64` and `RFAcc-64-P` could further double the energy savings.

To better analyze the energy-efficiency of RFAcc, Figure 13 shows the average power during an entire training of RF. Although poker and covtype on GPU consume less energy than CPU, the power of GPU is still as mush as twice higher than that of CPU. `RFAcc`'s power is only less than 1.04% of that of GPU. `RFAcc-64` slightly increases power to 2.1% due to more cells are read simultaneously during comparison. However, when node parellel optimization is enabled, `RFAcc-P` and `RFAcc-64-P` consumes more power (5.2%) due to more RCUs are activated at the same time.

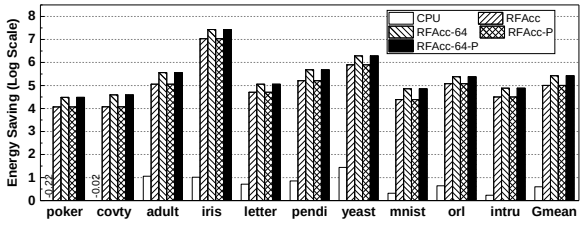


Figure 12: Energy savings over GPU.

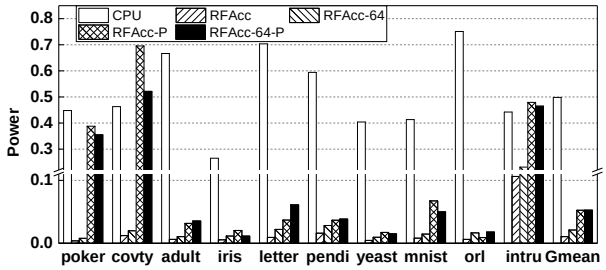


Figure 13: Power normalized to GPU.

7.4 Unary Encoding Optimization

We then evaluated unary encoding optimization. Figure 14 and Figure 15 report the speedup and energy savings, respectively, when

adopting different unary encoding configurations. All experiments are enabled with node parallel optimization at the same time.

Figure 14 shows that an X-unary encoding with larger X achieves better performance as each value comparison takes fewer cycles to finish. However, an X-unary encoding with larger X demands more ReRAM space. For example, 4-unary encoding demand 2 times space then no-encoding, while 64-unary encoding demands 8 times space. etc. As X grows, the average speedups are 1615×, 3225×, 4685×, 6432×, 7342× and 8564×, respectively.

As shown in Figure 15, unary encoding reduces energy consumption. This is because RFAcc with encoding needs significantly less execution time. For example, the energy savings increases from 10^5 with no encoding to 2.6×10^5 with 64-unary encoding. However, as shown by RFAcc and RFAcc-64 bars in Figure 13, the power of encoding is actually higher than that of no-encoding scheme since more cells are activated simultaneously.

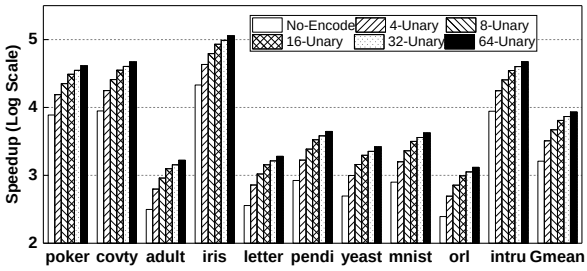


Figure 14: Comparing speedups with unary encoding.

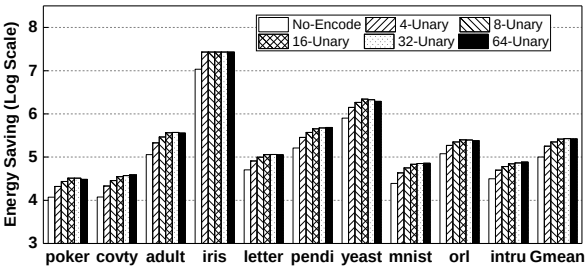


Figure 15: Comparing energy savings with unary encoding.

7.5 Impact of Array Size

Finally, we studied the impact when employing different dimensions of 3D ReRAM arrays. The average speedup and energy saving results are summarized in Figure 17 and Figure 18, respectively. All the experiments are on RFAcc with no encoding and node parallel optimizations. The figures show that 3D ReRAM array has an important impact on the overall performance and energy savings.

From Figure 17, when adopting larger 3D ReRAM arrays (i.e., more layers and larger array sizes), RFAcc could achieve better performance. To better understand this, Figure 16 compares the access time (extracted from NVSIM) with different numbers of layers and array sizes. More layers also increases energy saving (as shown in Figure 18) thanks to the lower per-bit search power [19].

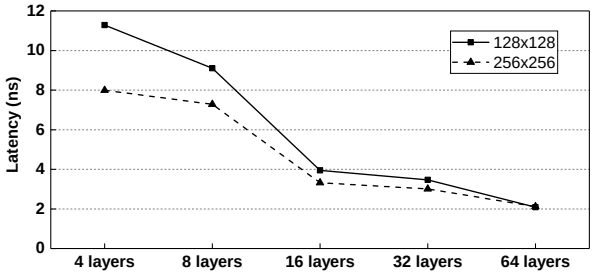


Figure 16: The access latency for arrays with different layers.

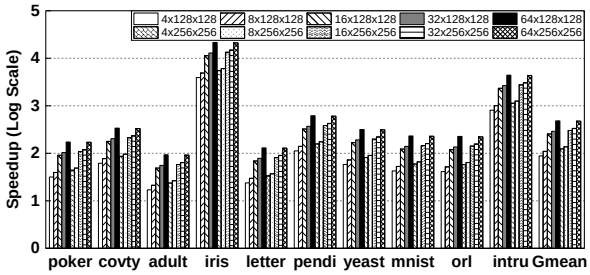


Figure 17: Comparing speedups with different array size.

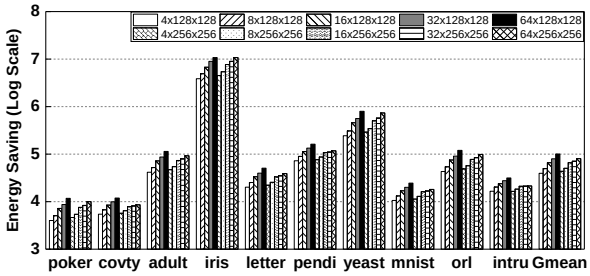


Figure 18: Comparing energy savings with different array size.

8 CONCLUSION

In this paper, we proposed RFAcc, a 3D ReRAM based PIM accelerator, to speedup random forest training. The novel relational comparator devised in this paper is the first in the literature. By eliminating data movement and enabling concurrent value comparisons, RFAcc outperforms over both CPU and GPU implementations. The three proposed optimizations further exploits the potential parallelism to greatly improve training performance and achieve significant energy consumption reductions over CPU and GPU implementations.

REFERENCES

[1] Mahdi Nazm Bojnordi and Engin Ipek. 2016. Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *International Symposium on High Performance Computer Architecture*.

[2] Leo Breiman. 2001. Random forests. *Machine learning* (2001).

[3] Geoffrey W Burr, Robert M Shelby, Severin Sidler, Carmelo Di Nolfo, Junwoo Jang, Irem Boybat, Rohit S Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U Giacometti, et al. 2015. Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element. *IEEE Transactions on Electron Devices* (2015).

[4] Chuan Cheng and Christos-Savvas Bouganis. 2013. Accelerating random forest training process using FPGA. In *International Conference on Field programmable Logic and Applications*.

[5] Christophe J Chevallier, Chang Hua Siau, Seow Fong Lim, Sri Rama Namala, Misako Matsuoka, Bruce L Bateman, and Darrell Rinerson. 2010. A 0.13 μm 64Mb multi-layered conductive metal-oxide memory. In *International Solid-State Circuits Conference*.

[6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *International Symposium on Computer Architecture*.

[7] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* (1995).

[8] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *International Symposium on Low-Power Electronics and Design*.

[9] Yexin Deng, Hong-Yu Chen, Bin Gao, Shimeng Yu, Shih-Chieh Wu, Liang Zhao, Bing Chen, Zizhen Jiang, Xiaoyan Liu, Tuo-Hung Hou, et al. 2013. Design and optimization methodology for 3D RRAM arrays. In *International Electron Devices Meeting*.

[10] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. 2012. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2012).

[11] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research* (2014).

[12] Håkan Grahn, Niklas Lavesson, Mikael Hellborg Lapajne, and Daniel Slat. 2011. CudaRF: a CUDA-based implementation of random forests. In *IEEE/ACS International Conference on Computer Systems and Applications*.

[13] Li-Yue Huang, Meng-Fan Chang, Ching-Hao Chuang, Chia-Chen Kuo, Chien-Fu Chen, Geng-Hau Yang, Hsiang-Jen Tsai, Tien-Fu Chen, Shyh-Shyuan Sheu, Keng-Li Su, et al. 2014. ReRAM-based 4T2R nonvolatile TCAM with 7x NVM-stress reduction, and 4x improvement in speed-wordlength-capacity for normally-off instant-on filter-based search engines used in big-data processing. In *Symposium on VLSI Circuits Digest of Technical Papers*.

[14] Wenqin Huangfu, Shuangchen Li, Xing Hu, and Yuan Xie. 2018. RADAR: a 3D-reRAM based DNA alignment accelerator architecture. In *Design Automation Conference*.

[15] Kaggle. 2019. Kaggle Competitions. https://www.kaggle.com/. (2019).

[16] Subhash Kak. 2016. Generalized unary coding. *Circuits, Systems, and Signal Processing* (2016).

[17] Wang Kang, Haotian Wang, Zhaohao Wang, Youguang Zhang, and Weisheng Zhao. 2017. In-memory processing paradigm for bitwise logic operations in STT–MRAM. *IEEE Transactions on Magnetics* (2017).

[18] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998).

[19] Shuangchen Li, Liu Liu, Peng Gu, Cong Xu, and Yuan Xie. 2016. Nvsim-cam: a circuit-level simulator for emerging nonvolatile memory based content-addressable memory. In *International Conference on Computer-Aided Design*.

[20] Yisheng Liao, Alex Rubinsteyn, Russell Power, and Jinyang Li. 2013. Learning random forests on the GPU. *New York University, Department of Computer Science* (2013).

[21] M. Lichman. 2013. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml. (2013).

[22] Nvidia. 2019. Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface. (2019).

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011).

[24] Robert E Schapire. 1990. The strength of weak learnability. *Machine learning* (1990).

[25] Hannes Schulz, Benedikt Waldvogel, Rasha Sheikh, and Sven Behnke. 2015. CURFIL: Random Forests for Image Labeling on GPU.. In *International Conference on Computer Vision, Theory and Applications*.

[26] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *International Symposium on Computer Architecture*.

[27] Mrigank Sharad, Charles Augustine, Georgios Panagopoulos, and Kaushik Roy. 2012. Spin-based neuron model with domain-wall magnets as synapse. *IEEE Transactions on Nanotechnology* (2012).

[28] Cong Xu, Dimin Niu, Shimeng Yu, and Yuan Xie. 2014. Modeling and design analysis of 3D vertical resistive memoryâĂŤA low cost cross-point architecture. In *Asia and South Pacific design automation conference*.

[29] He Zhao, Graham J Williams, and Joshua Zhexue Huang. 2017. wsrf: An R Package for Classification with Scalable Weighted Subspace Random Forests. *Journal of Statistical Software* (2017).

[30] Ji Feng Zhi-Hua Zhou. 2017. Deep Forest: Towards An Alternative to Deep Neural Networks. In *International Joint Conference on Artificial Intelligence*.