2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 978-1-6654-4507-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCAD51958.2021.9643480

Massively Parallel Big Data Classification on a Programmable Processing In-Memory Architecture

Yeseong Kim
DGIST
yeseongkim@dgist.ac.kr

Mohsen Imani
UC Irvine
m.imani@uci.edu

Saransh Gupta
UC San Diego
sgupta@ucsd.edu

Minxuan Zhou
UC San Diego
miz087@ucsd.edu

Tajana S. Rosing
UC San Diego
tajana@ucsd.edu

***Abstract*—With the emergence of Internet of Things, massive data created in the world pose huge technical challenges for efficient processing. Processing in-memory (PIM) technology has been widely investigated to overcome expensive data movements between processors and memory blocks. However, existing PIM designs incur large area overhead to enable computing capability via additional near-data processing cores and analog/mixed signal circuits. In this paper, we propose a new massively-parallel processing in-memory (PIM) architecture, called CHOIR, based on emerging nonvolatile memory technology for big data classification. Unlike existing PIM designs which demand large analog/mixed signal circuits, we support the parallel PIM instructions for conditional and arithmetic operations in an area-efficient way. As a result, the classification solution performs both training and testing on the PIM architecture by fully utilizing the massive parallelism. Our design significantly improves the performance and energy efficiency of the classification tasks by 123× and 52× respectively as compared to the state-of-the-art tree boosting library running on GPU.**

I. Introduction

We live in a world where technological advances are continually creating more data than what we can cope with. An attractive solution is to enable memory blocks to perform computations. This approach, often called processing in-memory (PIM), reduces communication costs between the processor and memory by performing most computations directly in memory [1]–[3]. Emerging non-volatile memory (NVM) technology, e.g., resistive memory (memristor or ReRAM), has been widely used for the PIM-based designs due to its high efficiency such as high density and low-power consumption.

To enable the computing capability to where the data are placed, prior research mainly investigated two architectural solutions, (i) near-data computing (also known as logic-in-memory, Figure 1a) and memory-based processor (Figure 1b). The memory-oriented architectures illustrate an intriguing future for next-generation computing paradigms; the design of efficient in-memory computing systems is still an open research question. The near-data computing [4]–[10] leverages the 2.5/3D-stacking technology to fabricate computing logic near to the existing memory. This approach can offer rich operations based on mature CMOS technology, and benefit high bandwidth between the logic and memory stack. However, the parallelism that the near-data computing can offer is typically restricted due to the high energy and area costs of the CMOS logic.

The second approach exploits the resistive memory to design a *memory-based processor* [3], [11]. These designs perform computation in-situ, i.e., inside memory bit lines, where the data are either stored in the resistive memory or converted to analog signals. This technique provides extremely high parallelism for arithmetic operations, e.g., vector-matrix multiplications, which are common on many applications such as deep learning. However, the limited functionality hinders supports of the wider variety of applications which require non-arithmetic operations, e.g., *conditional* operations. Furthermore, the memory-based processor needs large peripherals, e.g., analog-digital/digital-analog converters (ADCs/DACs) take the majority of the chip area around 65% [3]. Due to the area overhead, the memory-based processor utilizes conventional off-chip memory to store a considerable amount of data, potentially incurring high communication costs. Thereby, these technologies would not be appropriate solutions to incorporate computing capability into storage-class memory.

In this paper, we propose a novel processing in-memory system which allows pushing various computations onto the memory *itself* without expensive computing logic and accesses to off-chip memory, thus significantly reducing inter-component bandwidths (Figure 1c). To support various and generic in-memory operations, we exploit content addressable memory (CAM) structure [12] and in-memory NOR operation [13] as the basic blocks in tandem. Both techniques are implemented with low-overhead logic instead of using costly ADCs/DACs. Thus, we can serve *storage-class* density which is essential for the big data analysis applications. Our PIM instructions are designed in a similar fashion to the conventional instructions of general-purpose processors, e.g., condition flags and single instruction multiple data (SIMD) operations; but also offer *massively parallel* PIM executions. For example, the proposed architecture can execute up to 36K conditional operations (e.g., less than) and 12M arithmetic operations (e.g., addition) in parallel.

To present the practical value of the proposed architecture, we also show how to rewrite *non-arithmetic-centric* machine learning (ML) applications on our system supporting a programmable interface. With a given dataset stored in our memory-based architecture, our rewritten software orchestrate three *in-situ* data analysis procedures, genetic algorithm, decision tree learning, and adaptive boosting (AdaBoost) [14]. The outcome of the learning procedure is a tree boosting model which has been widely used in the field of big data mining [15]. Running the tree boosting on the memory-based processor is challenging since it involves frequent conditional executions unlike deep learning. Utilizing the massive parallel conditional and arithmetic operations, we can perform both the model training and inference tasks on where the data are actually stored.

In this paper, we make following contributions:

1) We propose a programmable NVM-based PIM architecture with a big data classification solution, called Classification Hypothesis Online Identifier on Resistive memory (CHOIR). The proposed architecture enables in-memory computing capability only using low-overhead circuits, thus suitable for the storage-class memory.

2) We propose a novel PIM instruction set architecture (ISA) which processes both *conditional* and *arithmetic* operations in a *massive-parallel*

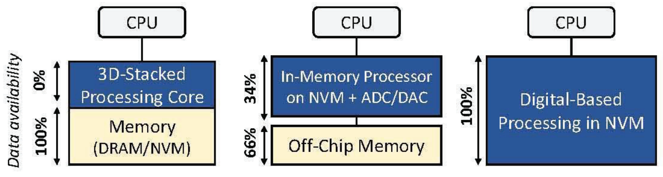


Fig. 1. Comparison of Near-/In-Memory Computing. The data availability for (b) is estimated for the design shown in [3]

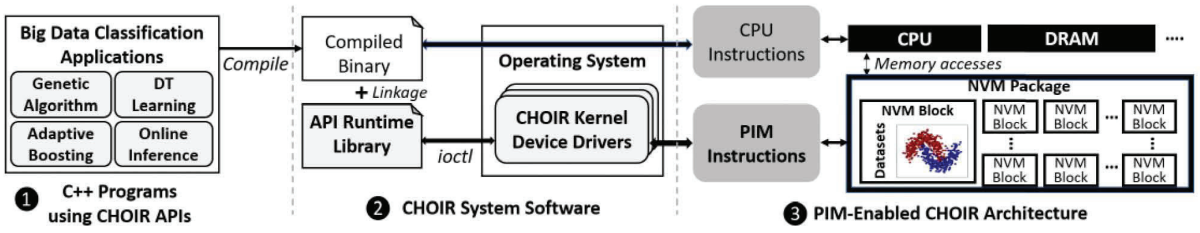


Fig. 2. Overview of CHOIR Architecture

way. To the best of our knowledge, it is the first design that supports the massive conditional operations in the ReRAM PIM architecture.

3) We show a comprehensive system software stack, which includes programming model, runtime library, and system-level management, to interface user programs with the underlying PIM hardware.

4) We propose an efficient and accurate learning algorithm for the tree boosting model. The training and inference procedures are fully parallelized on the proposed architecture.

Our experimental results show that CHOIR accurately classifies diverse data of practical applications that include millions of classification samples, e.g., 98.1% for motion tracking and 97.2% for fetal disease diagnosis, which are also comparable to the state-of-the-art learning techniques including DNN. Our design achieves high energy and performance efficiency by 123×/52× for training as compared to the state-of-the-art GPU running XGBoost [15]. In addition, the proposed design can be implemented with minimal area overhead of 4% on the existing resistive memory technololgy.

II. Overview of CHOIR Architecture

This work seeks to devise a highly-efficient architectural solution that supports sufficient programmability for big data classification. Figure 2 illustrates the overview of the CHOIR architecture consisting of three components: ❶ user-level programs implemented with CHOIR APIs, ❷ system software, and ❸ PIM-enabled architecture.

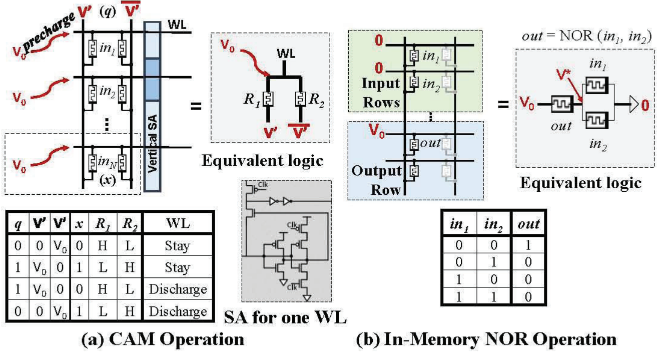
The proposed architecture can accelerate various programs implemented with a CHOIR software library. The program views the memory blocks as a list of 2D arrays, and processes the data using the PIM APIs such as *less-than* and *addition* functions. This abstraction is similar to popular scientific computing frameworks, e.g., Numpy array [16] and TensorFlow Tensor [17]. The API calls are linked with the CHOIR runtime library which executes PIM instructions through the kernel device drivers using `ioctl` system calls.

The CHOIR architecture has a host CPU and a storage-class memory system, called *NVM package*, consisting of multiple NVM blocks. The CPU can access the NVM package as a conventional memory subsystem like PCIe SSD. The PIM instructions are executed *inside* each NVM block which stores the processed data, e.g., training datasets. It offers massive parallelism by utilizing the large number of memory blocks as multiprocessors. For example, multiple blocks can execute the same PIM instruction in parallel. Furthermore, the NVM block can also execute a PIM instruction in a row/column parallel way, e.g., executing a conditional operation for all rows.

A. Background: Building Blocks of CHOIR

Before describing our CHOIR architecture, we here discuss which in-memory computing mechanisms are used to build the PIM functionalities.

Content addressable memory (CAM) The first building block is the CAM structure originally used for finding a target value inside memory. The conventional CMOS-based CAM structure has been utilized in limited applications, e.g., network routers, due to its significant power



q	V'	V'	x	R_1	R_2	WL
0	0	V_0	0	H	L	Stay
1	V_0	0	1	L	H	Stay
1	V_0	0	0	H	L	Discharge
0	0	V_0	1	L	H	Discharge

in_1	in_2	out
0	0	1
0	1	0
1	0	0
1	1	0

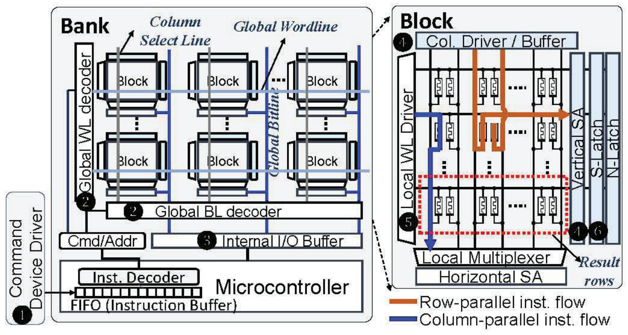
Fig. 3. Building Blocks for In-Memory Computations

consumption; recent research efforts show that we can implement efficient CAM logics in resistive memory. In particular, we exploit a crossbar-compatible design shown in [12].

We abstract the CAM functionality as a *parallel bit search mechanism* for entire memory rows. A single memory block can be viewed a 2D array whose element is a single bit. For an array column, we can identify which rows store the same values to a *query bit*, q.

Figure 3a describes the detailed CAM mechanism. In the initial stage, we precharge the horizontal wordline (WL) for the rows with V_0. The q is encoded by the bitline (BL) input voltages, V' and $\overline{V'}$. As shown in the table of Fig. 3(a), if q is '0', V' and $\overline{V'}$ are set to 0 and V_0; otherwise, they are set to V_0 and 0. If q is not the same to the value x stored in a row, the potential quickly discharges. In the opposite case, it stays to the high voltage level since the high resistance (H) keeps V_0. Thus, by checking the voltage level at the vertical sense amplifier (SA), we can select the desired rows. We design row-parallel operations using the CAM functionality. For example, we implement the 'less-than' operation by comparing multi-row values bit by bit. We show the detailed operation mechanism of PIM operations in Section III-B.

In-memory NOR (MAGIC) The second building block is in-memory NOR operation, also known as MAGIC (Memristor-aided logic) [13], [18], [19], which can perform on any memristor devices without any extra circuits. The MAGIC operation is processed in a column-parallel way, i.e., we can compute the NOR logic for all columns located on any two rows. Figure 3b shows the details of the MAGIC operation. In the initial stage, the cells in the target output row are reset into the mode of low resistance (L). The local WL driver, in turn, grounds two input rows and activates an output row with V_0. Then, the voltage difference $\Delta V = V_0 - V^*$ is determined by the resistance of the input memristor cells, as shown in the equivalent logic. For example, when in_1 and in_2 are both '0', the cells are encoded with high resistance (H), and thus ΔV is a small value. In contrast, any of in_1 and in_2 has low resistance, ΔV is relatively high voltage, incurring switching activity for the output cell.



(a) Bank (b) Block

Fig. 4. CHOIR Memory Microarchitecture

In our design, we exploit the MAGIC operation to design column-parallel PIM operations. Since NOR is a universal gate, it allows implementing any boolean logic in a column-parallel way. For example, we implement the addition and multiplication by repeatedly computing the NOR logics for two rows.

III. CHOIR PIM Architecture

A. PIM-Enabled Memory Microarchitecture

Figure 4 illustrates an overview of the memory architecture in CHOIR. The NVM package consists of multiple banks, where a bank stores 512MB in our configuration. Each bank has multiple blocks which execute various PIM operations for the stored data. A bank shown in Figure 4a is responsible for translating PIM instructions, distributes the activation signals across multiple blocks, and manages inter-block communications for specific operations. A device driver, called command device driver, initiates the PIM instructions into a FIFO of the microcontroller (❶). The microcontroller decodes the instruction and initializes the global wordline (WL) and bitline (BL) decoders to activate the target memory blocks and cells (❷). The I/O buffer is used when the instruction requires inter-block communications (❸). The single memory block shown in Figure 4b provides parallelism for the two types of PIM instructions. (i) *row-parallel* instructions: the column driver and vertical sense amplifier (SA) implement them using the CAM functionality (❹). (ii) *column-parallel* instructions: the local WL driver processes them based on NOR operations (❺). The conditional flags are implemented with two latches which hold the digital signal. We called the two latches as *S latch* (selection latch) and *N latch* (non-selection latch) (❻). The memory block has a 2D array structure of R rows and C columns. To exploit the CAM functionality, a single bit is stored into a pair of cells using the two-bit encoding. In the execution of the supported PIM operations, the data in a single row are supposed to be aligned in 64-bit representation. For example, a pair of the row index and column index, (r, c), indicates the 64-bit value located at the $(c \times 64)$-th bit in the r-th row. In the rest of the paper, we call a 64-bit value as a *word*. The data can be read/written in the same way to the usual memory devices. In particular, the last row of each memory block is reserved to store the output of the PIM instructions.

B. Instructions Set Architecture

Based on the two building blocks, the proposed NVM-based PIM instructions process multiple data with a single instruction similar to SIMD. All operations can be executed across multiple blocks in parallel. Some operations can be executed in row/column-parallel ways as well, e.g., (i) comparing multi-row values with a target value and (ii) adding values for multiple columns simultaneously. To support the massive parallel execution, the instructions should have a large size of operands which specify ***nonconsecutive, unfixed*** indexes of rows/columns/blocks. It does not fit into the fixed-length instructions of the CPU architecture. For this reason, the program issues the PIM instructions directly into the NVM package, unlike the existing ISA designs for near-data processors which add new instructions in the CPU architecture [20].

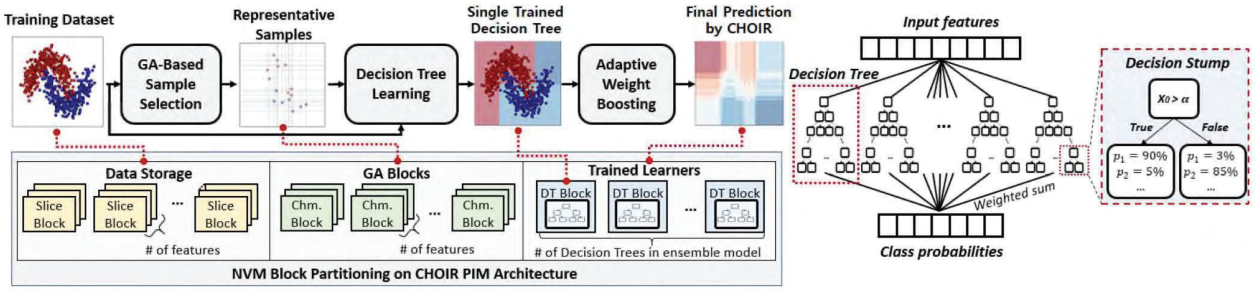
A unique property of our ISA design is that we loosely separate the data selection and computing instructions, whereas an instruction of the traditional ISAs typically includes both computing operation and target data locations (e.g., registers and addresses). For example, before executing a row-parallel comparison, it should specify target rows using a separate instruction. This design significantly avoids redundant communication between processors and memory, e.g., hundreds of bits indicating the target rows, since we do not need to send the operands for the data selection again when processing consecutive PIM operations with the same operands.

We issue a PIM instruction through a device driver by writing data to the instruction FIFO in the following format:

64 bit	64 bit		64 bit	8kB
Opcode	*1st operand*	···	*4th operand*	*Extended operand*

An instruction has a predefined opcode and up to four 64-bit operands. The *extended operand*, E, is used to specify a long bit stream for bitmask operands of column, row, and block indexes. The instruction set architecture (ISA) consists of multiple PIM instructions.

Data Selection Instructions (1) rgst i, E: The rgst instruction registers a list of block addresses, which are stored in E as a bitmask, to an integer identifier (ID) i. This instruction is called to map multiple blocks to the CHOIR array in the user-level program. In our implementation, the microcontroller of the NVM package stores the block lists mapped to each blockset ID. Note that the program usually needs to perform the registration only once during the initialization. Once the ID is registered, the program can perform multiple PIM instructions for the designated blocks by specifying the ID, instead of passing the long mask streams with the extended operand. (2-5) row/col i, r and rowmask/colmask i, E: The row/col selects a target row/column for the given index (r) of the blocks registered with i. For rowmask and colmask, E specifies bitmasks of multiple rows or columns. (6-7) sassoc/nassoc i: These instructions select new rows with the per-row conditional flags stored in either S latch or N latch. The local WL driver processes these instructions.
Row-Parallel Instructions: (8) incl i, v: The incl instruction is invoked with an operand, v, to perform an 'increment by 2^v' operation for the activated rows. Using the CAM functionality, we invert the v^{th} bits in two phases: i) writing 0s for the rows whose bits are 1s, and ii) write 1s for the opposite case. We then process the upper bits, i.e., $v+1^{th}$ bits, by iteratively repeating the same procedure for the rows whose previously bits were 1s. This stops when either there is no row to proceed or the most significant bits are inverted. (9) lt i, v: The lt instruction implements the typical 'less than' operation in a row-parallel way. We can identify the rows whose values are less than v for the selected rows. We implement this instruction by comparing each bit of v with all stored values. Based on the CAM functionality, we search which rows have zero values, starting from the most significant bit. If the bit of the same index in v is '1', the searched rows have value less than v, and thus we store them into the latches. In the opposite case (the bit in v is '0',) the search continues on the next bit of v. It is proceeded until the last bit is compared. As the final results, the *S latch* keeps the identified rows, while the *N latch* stores the rest of rows. (10) nsearch i, v: This instruction identifies the row that has the nearest value to the query value v based on the nearest(approximate)-CAM design proposed in [12], [21]. The identified rows are stored into the latches in the same way to lt. Note that, after executing lt and nsearch, the program can implement a 'if-condition-then-else' statement with sassoc and nassoc at the per-row granularity.



(a) Overview of CHOIR Classification (b) Tree Boosting Model

Fig. 5. CHOIR Learning Software on the PIM Architecture

Column-Parallel Instructions: (11-12) add/mul i: The add and mul instructions perform NOR-based addition and multiplication for the selected rows. We adopt the in-memory integer addition and multiplication mechanism shown in [13], [22]; floating-point data can be also processed in the same memory architecture [11]. In our design, we further parallelize the column-parallel operations in a *semi* row-parallel way. We divide a memory block into subarrays where they are connected with switches. The subarray can then perform NOR either individually or across subarrays by controlling the switch.

IV. Classification on CHOIR

A. Overview of CHOIR Classification

The software part of CHOIR is a supervised learning algorithm rewritten with the PIM APIs. It performs both training and inference procedures on the proposed architecture. Figure 5a shows an overview of the training procedure with the memory block partitioning which is mapped to the list of 2D arrays in the program procedure. The training dataset is stored in a set of blocks, called *data storage*. In this paper, we denote N for the number of samples, F for the number of features, and K for the number of classes, in the training dataset. The data storage stores the training data into multiple memory blocks. We call a single block as *slice block*. The proposed algorithm is designed to work with 64-bit integers since the floating-point operations take a larger number of cycles [11]. In the slice blocks, the values in a feature dimension are normalized to a range of $[0, I_{max}]$, where I_{max} is a large-enough integer.

The training procedure is performed by an interplay of three algorithms: genetic algorithm (GA), decision tree (DT) learning, and AdaBoost. Through an iterative training procedure on the PIM architecture, the CHOIR software generates an ensemble of multiple decision trees, and all the decision trees involve the class inference. In the first training step, based on the genetic algorithm, it identifies a small number of samples, which represent the per-feature distribution of the training data (Section IV-B). The algorithm is implemented with the memory blocks called Chromosome blocks, in short *chm blocks*. With the selected samples, the second step performs the decision tree learning (Section IV-C). In this step, most computations happen inside the data storage, and thus the processor does not need to access the training dataset directly. The trained DT is written to a memory block called *DT block*. As the last step, we exploit *AdaBoost*, which is known as one of the best off-the-shelf ML algorithms [23] (Section IV-D).

Figure 5b illustrates the tree boosting model. The AdaBoost utilizes many simple, complementary learning models, called either *weak/base learners*, to create a strong prediction. A decision tree (DT) comprises multiple decision nodes, also known as *decision stump* (DS). We construct a DT by connecting multiple decision stumps, so that it supports multi-level decisions by up to a configurable level, H. A DS is defined with a feature, f, and a decision rule value, v^f, where the decision is made by comparing the feature value of a sample, i.e., $v^f < x_i^f$. It has two branches connected to either another stump or a leaf node. Each leaf node includes the probability values for each class, i.e., $\mathbf{p} = \langle p_0, \cdots, p_{K-1}\rangle$. The prediction results of all decision trees are combined with weighted sums, creating the final prediction results of class probabilities (Section IV-E).

B. GA-Based Sample Selection

Our GA procedure identifies the representative subset of N_{chm} values for each feature. There are two types of the *chm* blocks, *parent* and *child chm blocks*, where $C/2$ blocks are allocated for each type to store multiple subset candidates. A parent *chm* block is initialized with N_{chm} feature values of random samples. For each iteration, it first calculates a ***fitness metric***, i.e., how much the selected values in each *parent chm block* mimic the distribution of the original training data. We use mean absolute deviation (MAD) as our fitness metric:

$$\{\boldsymbol{\tau}^f(d) | \boldsymbol{\tau}^f(d) = \frac{\sum_i^N |t_d - x_i^f|}{N}, t_d = \frac{(d+1)\cdot I_{max}}{D+1}\}$$

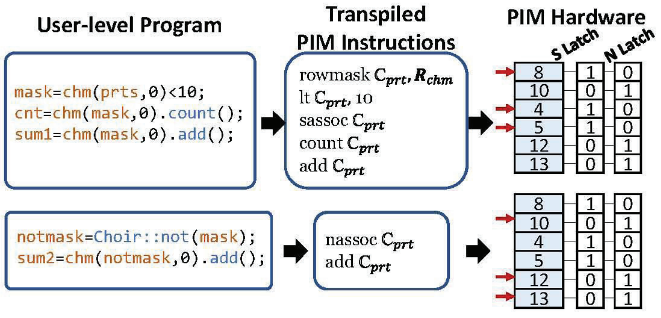
where x_i^f is one of N features and D is a predefined value that decides the number of partitions. It is decomposed as follows:

$$\boldsymbol{\tau}^f(d) \times N = t_d \cdot n_c^f - \sum_{x_i < t_d} x_i + \sum_{x_i \geq t_d} x_i - t_d \cdot (N - n_c^f)$$

where n_c^f is the number of rows whose value x_i are smaller than t_d. Since all the terms in the equation can be computed in parallel using the PIM operations, i.e., less-than, addition, and count operations, the CHOIR architecture can efficiently perform the GA selection procedure.

Figure 6 illustrates how to compute this procedure with CHOIR PIM APIs. In this example, for the memory-mapped CHOIR array (chm variable), the user program specifies the target rows and columns to execute the PIM operations. The traspiled instructions first find the rows whose values is smaller than t_d (in this example, $t_d = 10$), and count n_c^f. Then, with the two addition operations and latched results, it computes the two partial sums, i.e., $\sum_{x_i < t_d} x_i$ and $\sum_{x_i \geq t_d} x_i$. The host processor can calculate the MAD metric using the three variables computed inside memory, sum1, sum2 and cnt.

We then update the *child chm blocks* by randomly selecting two parent blocks and mixing their row values. The updated children should be more similar to the original distribution. To this end, we exploit the fitness proportionate selection metric so that the parent blocks with smaller distribution differences get higher chance to be chosen. While the cross-block copy command writes most rows of the child block, i.e., copy, a very small number of rows are updated with samples of the training dataset to encourage diversity of explored sample combinations. The percentage of rows updated from the training dataset is determined by a mutation probability, $p_{\boldsymbol{\mu}}$. It is executed until all the child *chm* blocks are



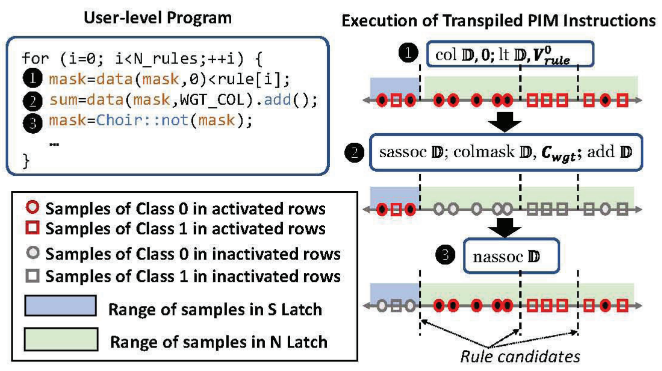
```
mask=chm(prts,0)<10;
cnt=chm(mask,0).count();
sum1=chm(mask,0).add();
```

rowmask $\mathbb{C}_{prt}, \mathbb{R}_{chm}$
lt $\mathbb{C}_{prt}$, 10
sassoc $\mathbb{C}_{prt}$
count $\mathbb{C}_{prt}$
add $\mathbb{C}_{prt}$

```
notmask=Choir::not(mask);
sum2=chm(notmask,0).add();
```

nassoc $\mathbb{C}_{prt}$
add $\mathbb{C}_{prt}$

Fig. 6. GA-based Sample Selection ($\mathbb{C}_{prt}$: block set ID for parent *chm* blocks, $\mathbb{R}_{chm}$: bitmask for the first N_{chm} rows)


User-level Program
Execution of Transpiled PIM Instructions
Samples of Class 0 in activated rows
Samples of Class 1 in activated rows
Samples of Class 0 in inactivated rows
Samples of Class 1 in inactivated rows
Range of samples in S Latch
Range of samples in N Latch
Rule candidates

```
for (i=0; i<N_rules;++i) {
❶ mask=data(mask,0)<rule[i];
❷ sum=data(mask,WGT_COL).add();
❸ mask=Choir::not(mask);
   …
}
```

❶ col $\mathbb{D}, 0$; lt $\mathbb{D}, \boldsymbol{V}^0_{rule}$
❷ sassoc $\mathbb{D}$; colmask $\mathbb{D}$, $\boldsymbol{C}_{wgt}$; add $\mathbb{D}$
❸ nassoc $\mathbb{D}$

Fig. 7. Decision Stump Learning ($\mathbb{D}$: block set ID of slice blocks, $\mathbf{V}_{rule}$: decision rule candidates, $\mathbf{C}_{wgt}$: bitmask for the columns of sample weights)

updated. Through multiple iterations, this GA procedure can find a subset of the samples that has very similar distribution to the original one.

C. Decision Tree Training

The goal of the DT training is to learn each DS staring from the root node and write the trained stump in the DT memory block. To create a DS, this step evaluates multiple decision rule candidates using the best representative samples selected by the GA algorithm. For example, for a decision rule candidate, $\overline{v}^f$, the feature dimension is divided into two partitions. Since the boosting procedure gives weight values for each sample, we consider the weight values in this evaluation. Let us assume that W_k^L is the weight sum of all samples for each class, k, in the left partition and W_k^R is that of the right partition. Then, we can evaluate the classification quality using *Entropy* metric as $-\sum_k^K (p_k^L \cdot log(p_k^L) + p_i^R \cdot log(p_k^R))$, where $p_k^L = W_k^L / \sum_{k'} W_k^L$ and $p_k^R = W_k^R / \sum_{k'} W_k^R$. If the decision rule accurately partitions, this metric has a small entropy.

Figure 7 shows how we compute the weight sum to compute the entropy. This procedure is running on the slice blocks of the data storage (mapped to data variable). In the computation, we divide a feature dimension into multiple partitions using each rule candidate. To compute the entropy, the parallelized PIM commands compute the less than operation for the first partition (❶). It activates the samples in the left partition to compute the partial weight sum with add (❷). Then, it selects all the data samples, which are larger than the rule, by activating with the N latch (❸). By repeating it for each partition, we can compute weight sums of all partitions. The host processor then computes which rule has the minimum entropy and writes it into the DT block as the best decision rule. Lastly, with incl operation, we update the slice blocks to keep the index of each decision node, called *node ID*, which indicates where the sample is classified by the current DT.

D. Adaptive Weight Boosting

We create multiple DTs by boosting (increasing) the weights of weak hypothesis, i.e., inaccurately classified samples, while setting smaller weights for the correctly classified samples. The next DT is created using the boosted weights, and the accuracy can either increase or converge with more base learners. We utilize a state-of-the-art boosting mechanism, SAMME.R which supports multi-class classification [14]. SAMME.R boosts each sample weight if the samples predicted incorrectly. This algorithm always converges as the SAMME.R mechanism guarantees it as long as each DT makes the prediction better than random guessing. After boosting the sample weights, we re-normalize the weight values to map them into the 64-bit integer range that the PIM can support, while selecting the N_{chm} highest weight samples for the DT learning procedure. Then, we update the slice blocks using the calculate weights and reset the node ID for each sample. To update the values across multiple slice blocks of different features, this operation is performed in a block-parallel way using cwrite. Lastly, we update the probability values of leaf nodes, so that each DT has the same importance in the final ensemble model. Through the iterations, the CHOIR software creates M decision trees and produces the ensemble model into the DT blocks.

E. Online Class Inference

The online inference procedure is very similar to the decision stump training. We classify multiple samples at the same time by performing the row-parallel comparisons and then updating node IDs. In the initial step, we execute ltd to compare the rule values with different features of the slice blocks. For the selected rows, we update the node IDs of each sample by executing csassoc and incld. After computing all the decision rules, we can identify the probability of the node ID for each sample by reading the stored weights from the DT blocks, i.e., $\mathbf{p}^m_{\boldsymbol{\theta}} = \langle p_{\boldsymbol{\theta},0}, \cdots, p_{\boldsymbol{\theta},\boldsymbol{K}-1} \rangle$ where the classified node ID is $\boldsymbol{\theta}$. The final classification result for each sample is the class that has the highest probability sum, i.e., $\underset{k}{\operatorname{argmax}} \sum_{m=0}^{M-1} \mathbf{p}^m_{\boldsymbol{\theta},\boldsymbol{k}}$.

V. Evaluation

A. Experimental Setup

Table I summarizes the experimental setup and the parameters of CHOIR used in our evaluation. We implement CHOIR classification procedure using C++ with the CHOIR runtime. The host system runs on Intel i7-8700K CPU at 3.70GHz with 16 GB main memory. To evaluate the proposed architecture and software, we have developed an in-hour simulation infrastructure which estimates the performance and power of the CHOIR storage system in a cycle-accurate manner. We also develop a front-end of the simulator using the device drivers to i) emulate the PIM functionalities using GPU memory and CUDA threads on NVIDIA GTX 1080 Ti, and ii) record the executed PIM instructions. With the instruction execution log, we perform a circuit-level simulation to obtain the total cycles and energy on the PIM-enabled storage microarchitecture, including the memresistors, peripherals, memory blocks, IO buffer, and microcontroller of the bank level. We use HSPICE and Synopsys design compiler in 45nm TSMC technology. The proposed design work with any bipolar resistive technology; we use a single-level memristor device proposed in [24] with $R_{on} = 10K\Omega$ and $R_{off} = 10M\Omega$. We use the detailed VTEAM parameters reported in [13]. The synthesized circuit design produces the latency and energy for each PIM instruction during the storage-side execution. For the host side, we measure the power using HIOKI 3334 meter, and exclude the execution time and power consumed on the GPU used for the storage simulation.

We utilize 13 benchmarks summarized in Table II. We compare the CHOIR classification to a popular machine learning package, scikit-learn (sklearn), an OpenCL implementation of the CHOIR procedure

TABLE I
EXPERIMENTAL SETUP AND CHOIR PARAMETERS

Host Systems		**CHOIR**			
CPU	Intel(R) Core(TM) i7-6700K, 8 Cores @ 4.0 GHz	**Technology**	45nm TSMC	**Bank**	
CPU Cache	L1: 32kB, L2: 256kB, L3:8MB	**Memristor**	VTEAM [24]	**Size**	512 MB
Main memory	16 GB (DDR4) @ 2133 Mhz	**Block**		**Area**	69.98 mm^2
OS Kernel	Linux 4.15 (Ubuntu 18.04)	**Size**	128 kB	**Total**	
CHOIR Runtime Compiler	GCC 5.4 (option: -O3 -ftree-vectorize)	**Memresistor**	0.017 mm^2	**Size**	4 GB (8 Banks)
Bandwidth from/to CHOIR	12 GB/s (PCIe)	**Peripheral (SA+latch)**	509 μm^2	**Area**	559 mm^2

TABLE II
CLASSIFICATION BENCHMARKS

Name	F	K	N **(train)**	$\overline{N}$ **(inf.)**	**Description**
BLOBS	2	3	2048	2048	Microbenchmark [25]
IRIS	4	3	135	15	Flower pattern recognition [26]
SHUTTLE	9	8	43500	14500	**Airplane stat log classification [27]**
PAGE	10	5	4925	548	**Page block classification [28]**
CARDIO	21	3	1913	213	Cardiotocography [29]
PAMAP2	27	5	16384	16384	Activity recognition with IMU [30]
AUDIO	388	7	38513	2027	**Music identification (Audioset) [31]**
UCIHAR	561	12	6213	1554	Activity recognition with mobile [32]
ISOLET	617	26	6238	1559	Voice recognition [33]
MNIST	392	10	60000	10000	Text recognition [34]
FACE	608	2	22441	2494	Face recognition [35]
HIGGS	28	2	10000000	1000000	Physics simulation [36], [37]
SUSY	18	2	5000000	500000	Physics simulation [37], [38]

TABLE III
CLASSIFICATION ACCURACY COMPARISON

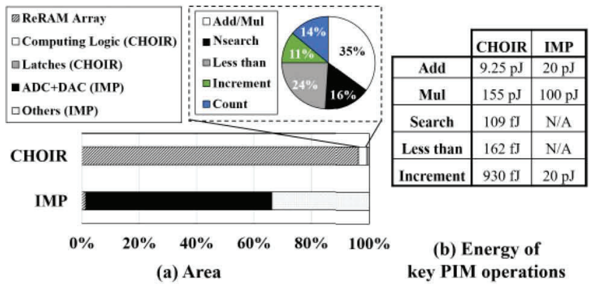
	H	**M**	**CHOIR**	**Sklearn**	**XGBoost**	**DNN**
BLOBS	2	128	100.00%	100.00%	100.00%	100.00%
IRIS	3	128	100.00%	100.00%	100.00%	100.00%
SHUTTLE	3	128	99.94%	99.99%	99.99%	98.94%
PAGE	6	1024	97.08%	96.90%	97.26%	94.52%
CARDIO	3	1024	97.18%	95.77%	96.24%	94.96%
PAMAP2	3	1024	98.05%	96.56%	97.82%	92.98%
AUDIO	6	2048	59.34%	57.32%	58.70%	61.33 %
UCIHAR	6	2048	97.88%	96.55%	97.13%	96.6 %
ISOLET	6	2048	94.36%	91.78%	94.64%	95.6 %
MNIST	6	2048	97.78%	97.90%	98.71%	99.5 %
FACE	6	2048	98.71%	97.00%	97.43%	98.23 %
HIGGS	6	2048	81.55%	82.08%	82.33%	82.60%
SUSY	6	2048	80.29%	79.35%	81.20%	84.54%

that parallelizes tasks for each memory block using the GPU cores, and XGBoost which is a state-of-the-art boosting library fully parallelized on CUDA [15]. The parameters of the GA-based sample selectors are empirically set by $C = 32$, $N_{chm} = 64$ (128 for SUSY and HIGGS), $D = 8$, and $p_{\mu} = 0.005$.

We compare the CHOIR classification to a popular machine learning package, scikit-learn (sklearn) [25], an OpenCL implementation of the CHOIR procedure that parallelizes tasks for each memory block using the GPU cores, and XGBoost which is a state-of-the-art boosting library fully parallelized on CUDA [15].

B. Architectural Comparison

The key novelty of the proposed PIM architecture is that we perform the massive PIM operations without using large peripherals. Figure 8a shows the breakdown of area overhead for a memory block. We compare the CHOIR architecture with a state-of-the-art PIM design, IMP [3]. IMP performs parallel ReRAM-based computations similar to CHOIR; however using ADC/DAC units to enable the analog/mixed signal-based computation. The results show that the ReRAM array in CHOIR takes the majority of the area (96%), whereas a large area of IMP is taken by ADC/DAC units, i.e., 65%. In CHOIR, the computing logic including SA **modifications with CMOS circuits incurs only 3% area overhead, and the** latches take less than 1% of the total memory. The low-overhead CHOIR architecture can be a suitable solution for the storage-class memory.



	CHOIR	IMP
Add	9.25 pJ	20 pJ
Mul	155 pJ	100 pJ
Search	109 fJ	N/A
Less than	162 fJ	N/A
Increment	930 fJ	20 pJ

Fig. 8. Comparison of CHOIR with IMP

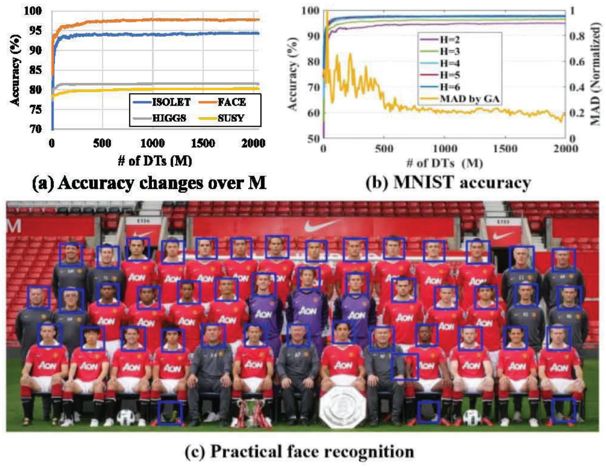
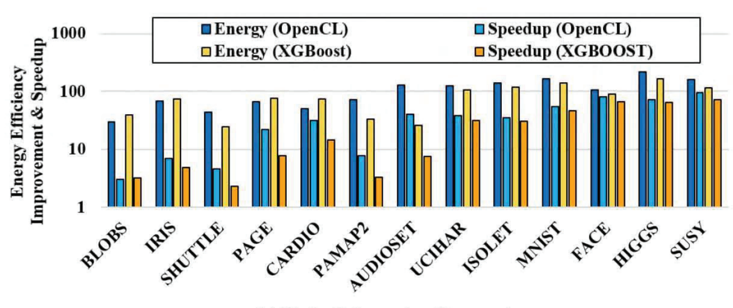


Fig. 9. Accuracy of Trained CHOIR Models

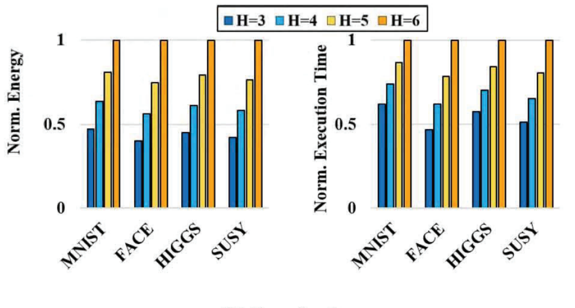
Figure 8b shows the breakdown of the area overhead to support each major operation: add/mul, less-than, search (and match), increment, and count. We observed that the two PIM functionalities that we newly propose, i.e., increment and less-than, can be implemented with comparable overhead to the other PIM operations. Figure 8c shows the comparison of energy consumption for the operations, which require to process a single word. The results show that, for the addition and multiplication operations, the energy consumption of CHOIR is comparable to the IMP design. The other CHOIR operations (search, match, and less-than), are performed with a negligible amount of energy. For the increment operation, we assume that IMP may support it using its addition operation. Since the CHOIR implements the operation by directly manipulating digital bits, it consumes less energy than IMP.

C. Classification Accuracy

Table III summarizes the classification accuracy for the inference datasets. It also shows the baseline model parameters, the height of a DT (H) and the number of DTs (M). The results present that the proposed



(a) Default Learning Parameter



(b) Tree depths

Fig. 10. Energy and Performance Improvement in Training

training algorithm can provide high accuracy for general classification problems. For example, we can achieve more than 97% for the diverse classification tasks including disease diagnosis (CARDIO), human activity recognition (PAMPA2, UCIHAR), and image recognition (MNIST, FACE). We also compare the accuracy with an AdaBoost implementation of the sklearn and XGBoost. The results show that CHOIR can provide comparable accuracy to the state-of-the-art ML libraries. As compared to the sklearn, CHOIR achieves better accuracy for eight benchmarks. This trend is typically observed for the benchmarks that include non-negligible noises, e.g., UCIHAR, ISOLET, and FACE.

Since the classification is made based on the representative samples selected by the GA-based algorithm, the CHOIR creates more robust models to the outliers. The CHOIR models also exhibit comparable accuracy to the DNN models. The results suggest that the proposed PIM-based learning solution can provide high quality for diverse applications which need frequent training, e.g, assimilation of data collected real-time.

Figure 9a shows how the accuracy changes over different numbers of DTs, M, for three representative benchmarks. The results show that the proposed algorithm successfully converts the base learners to the strong ensemble model. For example, the first DT for FACE predicts classes only with 84% accuracy, but by boosting the weight, it quickly achieves high accuracy, creating the final ensemble model that has 98% accuracy. Figure 9b describes the accuracy changes over different combinations of the two parameters for MNIST along with the MAD convergence by the GA. The result shows that we can achieve high accuracy with enough tree depths, e.g., more than 4. The GA also successfully optimizes the MAD cost function through iterations. Figure 9c presents the results of the face recognition, and the model created by CHOIR recognizes all 41 faces only with 4 false positives.

D. Energy and Performance Evaluation

Training Procedure We compare the energy and performance efficiency of the proposed CHOIR with the two GPGPU-based implementations. Figure 10a summarizes the comparison results of the training procedures when using the same training parameters shown in Table III. The proposed training procedure significantly improves the energy and performance efficiency. We observe higher improvements for large datasets since we can parallelize more operations while reducing communication costs to the host processor. We can achieve the energy and performance improvement on average by 153x and 63x compared to OpenCL-based CHOIR implementation, and 123x and 52x compared to XGBoost. These results show that the proposed CHOIR design which performs the tasks on the NVM-based PIM architecture can serve high efficiency for general classification problems.

The efficiency of the training procedure is also affected by the model complexity parameters. Figure 10b shows how energy and performance change for different tree depth settings. The results show that less-complex DTs can be trained with much less resources. Based on the trade-off between accuracy and efficiency, applications can make their strategies to select learning parameters. For example, for FACE, the accuracy difference between $M = 128$ and $M = 1024$ is only 2%, while training the model of $M = 128$ consumes 16x less energy as compared to the model of $M = 2048$.

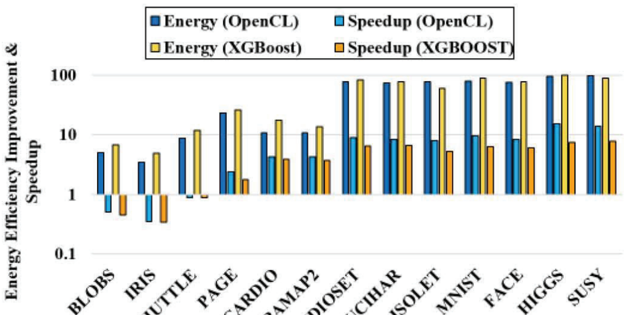


Fig. 11. Efficiency Improvement in Inference

Inference Procedure Figure 11 shows the energy and performance improvement compared to the GPU-based inference. The results show that CHOIR also efficiently performs the inference procedure. The efficiency improvement is less than that of the training procedure since the CPU needs to return the probability sums of all DTs. However, both training and inference involve many comparison operations which can be fully accelerated using the proposed row-parallel comparison operations. As the result, we can still improve the performance and energy on average by 83x (82x) and 11x (7x), respectively, as compared to the OpenCL-based implementation (XGBoost).

E. Energy and Performance Breakdown

Figure 12a and b compares the execution time breakdown of GPU and CHOIR architecture during the training procedure. For the GPU architecture, we used NVIDIA nvprof tool to extract the statistics. The result shows that the CHOIR significantly reduces the data movement cost as compared to the GPU architecture. For example, transferring the data between the main memory and the GPU memory (GDDR) takes 13.3%, and the GPU processing cores spend 70.9% of the total execution time to access the GDDR, in total 84.2%. In contrast, CHOIR reduces the data transfer cost by 93.6×.

The results also show that CHOIR performs the training procedure while highly utilizing the PIM-based computations. For example, the

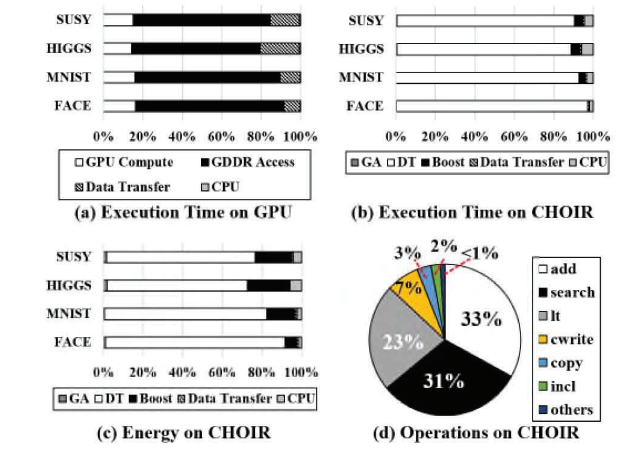


Fig. 12. Breakdown in Training

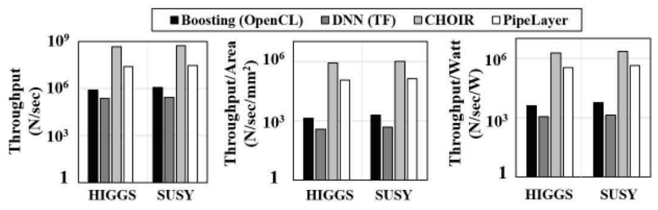


Fig. 13. Throughput Comparison

portion of the execution time running on the CPU is less than 10%. For the datasets that have the larger samples, e.g., HIGGS, the boosting procedure consumes more energy as shown in Figure 12c, since it needs more write operations to update sample weights. However, since the write operations are parallelized across blocks to hide the latency, there is no significant impact on performance. Figure 12d shows the breakdown of PIM operations used in the training procedure. The proposed algorithm can be executed with minimal cross-block data movements. For example, the cross-block write-related operations, e.g., cwrite and copy operations, only take 10% of the breakdown, while the other 90% of operations solely happen inside each memory block.

F. Throughput Comparison

Figure 13 compares the throughput of the big data classification for four different cases: i) tree boosting running on GPU, ii) DNN running on GPU with Tensorflow (TF) [17], iii) the proposed CHOIR design, and iv) a state-of-the-art PIM-based DNN accelerator, PipeLayer [2]. We use the two largest datasets, HIGGS and SUSY, which have 10 and 5 million data samples, respectively. The DNN model is trained with a 5-layer structure shown in [37], where each layer has 300 hidden units. We observe that the tree boosting is more light-weight than the sophisticated DNN. For example, the proposed algorithm is 5× faster than the TF-based DNN training even on GPU. As compared to the other PIM architecture, the throughput of CHOIR is 475M samples/sec, 18.7x higher than PipeLayer. When considering the total chip area, CHOIR also achieves higher computation efficiency in terms of throughput/area. The power efficiency of CHOIR is 1.83M (samples/sec/W), also higher than boosting on GPU (4.1K samples/sec/W), DNN on GPU (1.1K samples/sec/W), and Pipelayer (0.34M samples/sec/W).

VI. Related Work

Near-data computing accelerates computation by reducing the overhead of data movements between hardware blocks [4], [5], [9]. 3/2.5D stacking technologies are driving active recent research to integrate logic on die. For example, prior research designed PIM blocks for application acceleration, e.g., MapReduce based on 3D stacking [39], nearest neighbor search using computational logics beside DRAM [40], and parallel graph processing based on 3D DRAM [41]. Several designs have been also proposed for heterogeneous computing platforms to use the accelerators in system, offer high memory bandwidth, and coherently access host memory, e.g., IBM CAPI [42], Intel HARP [43], and HMC 2.0 [7], [20]. MICRON's IMI [10] exploits simple bit-serial CMOS computing elements to DRAM array's sense amplifiers. However, the execution of DRAM-integrated logics is destructive in that it invalidates the data used as operands, requiring many data writes after each parallel PIM operation [44]. In contrast, CHOIR performs massive-parallel and non-destructive operations inside NVM block without using high-cost CMOS logic, offering much higher efficiency to achieve the same level of parallelism.

Previous work have presented various PIM-enabled design [1], [45]–[48]. Work in [49] provided a survey with discussion for different NVM devices. Prior work also proposed a PIM-based design for DNN training acceleration based on specialized crossbar array [2], [50], [51]. Recent work utilized the in-memory computing to design an in-memory data parallel processor (IMP) [3]. In IMP, memory arrays act as vector processing units which perform arithmetic operations, e.g., addition and multiplication. However, their PIM operations are based on analog mixed-signal design which demand large area overhead. In contrast, CHOIR enables massive-parallel in-situ computation at near-zero silicon cost. A recent work [11], [52] proposed a custom PIM accelerator design for deep learning based on the MAGIC NOR technique and how to support floating-point operations with the in-memory NOR. In contrast, our work focus on how to design a programmable, cross-stack PIM architecture with the support of the in-memory conditional operations for non-arithmetic-centric ML workloads.

VII. Conclusion

In this paper, we propose a hardware/software codesign of CHOIR, which includes the new NVM-based architecture and PIM-friendly supervised learning algorithm. The CHOIR architecture supports PIM operations with minimal area overhead suitable for storage-class memory. The proposed classification algorithm can be implemented in a programmable way and performs both the model training and inference procedures efficiently on the PIM architecture. Our evaluation show that the proposed design have minimal area overhead of 4% on the existing resistive memory technology. The classification solution on CHOIR also accurately classifies practical applications which are also comparable to the state-of-the-art learning deep learning techniques. Our results show that the classification method on the PIM architecture achieves high energy and performance improvements as compared to GPU. For example, we improve the energy and performance efficiency by 123× and 52× respectively, for training as compared to the state-of-the-art GPU running XGBoost.

Acknowledgment

This work was supported by the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031). This work was also supported in part by National Science Foundation (NSF) #2127780 and #1730158, Semiconductor Research Corporation (SRC) Task No. 2988.001, CRISP (one of six centers in JUMP, an SRC program sponsored by DARPA,) Department of the Navy, Office of Naval Research, grant #N00014-21-1-2225, and a generous gift from Cisco.

References

[1] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 14–26.

[2] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 541–552.

[3] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 1–14.

[4] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[5] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[6] I. B. Peng and J. S. Vetter, "Siena: exploring the design space of heterogeneous memory systems," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 427–440.

[7] . M. Technology, "Hybrid memory cube," https://www.micron.com/products/hybrid-memory-cube, 2017.

[8] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 655–668.

[9] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin *et al.*, "Highly scalable near memory processing with migrating threads on the emu system architecture," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2016, pp. 2–9.

[10] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory intelligence," *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[11] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 802–815.

[12] M. Imani, A. Rahimi, and T. S. Rosing, "Resistive configurable associative memory for approximate computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 1327–1332.

[13] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, 2016.

[14] J. Zhu, H. Zou, S. Rosset, and T. Hastie, "Multi-class adaboost," *Statistics and its Interface*, vol. 2, no. 3, pp. 349–360, 2009.

[15] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[16] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, p. 22, 2011.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[18] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–7.

[19] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Proceedings of the 54imanth Annual Design Automation Conference 2017*. ACM, 2017, p. 6.

[20] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015, pp. 336–348.

[21] M. Imani, Y. Kim, and T. Rosing, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 757–763.

[22] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using magic," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 1–5.

[23] R. E. Schapire, "Explaining adaboost," in *Empirical inference*. Springer, 2013, pp. 37–52.

[24] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2015.

[25] "Scikit-learn machine learning in python," http://scikit-learn.org/stable/.

[26] "Uci machine learning repository, iris," https://archive.ics.uci.edu/ml/datasets/Iris.

[27] "Uci machine learning repository, statlog," https://archive.ics.uci.edu/ml/datasets/Statlog+%28Shuttle%29.

[28] "Uci machine learning repository, page," https://archive.ics.uci.edu/ml/datasets/Page+Blocks+Classification.

[29] "Uci machine learning repository, cardiotocography," https://archive.ics.uci.edu/ml/datasets/Cardiotocography.

[30] A. Reiss and D. Stricker, "Introducing a new benchmarked dataset for activity monitoring," in *Wearable Computers (ISWC), 2012 16th International Symposium on*. IEEE, 2012, pp. 108–109.

[31] J. F. Gemmeke, D. P. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter, "Audio set: An ontology and human-labeled dataset for audio events," in *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, 2017, pp. 776–780.

[32] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones." in *ESANN*, 2013.

[33] "Uci machine learning repository, isolet," https://archive.ics.uci.edu/ml/datasets/ISOLET.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[35] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results," http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[36] "Uci machine learning repository, higgs," https://archive.ics.uci.edu/ml/datasets/HIGGS.

[37] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, p. 4308, 2014.

[38] "Uci machine learning repository, susy," https://archive.ics.uci.edu/ml/datasets/SUSY.

[39] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 190–200.

[40] C. C. del Mundo, V. T. Lee, L. Ceze, and M. Oskin, "Ncam: Near-data processing for nearest neighbor search," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015, pp. 274–275.

[41] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015, pp. 105–117.

[42] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[43] P. K. Gupta, "Xeon+ fpga platform for the data center," in *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, vol. 119, 2015.

[44] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 288–301.

[45] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 27–39.

[46] T. Gokmen and Y. Vlasov, "Acceleration of deep neural network training with resistive cross-point devices," *arXiv preprint arXiv:1603.07341*, 2016.

[47] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 445–456.

[48] M. Imani, X. Yin, J. Messerly, S. Gupta, M. Niemier, X. S. Hu, and T. Rosing, "Searchd: A memory-centric hyperdimensional computing with stochastic training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2422–2433, 2019.

[49] G. W. Burr, R. M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola *et al.*, "Neuromorphic computing using non-volatile memory," *Advances in Physics: X*, vol. 2, no. 1, pp. 89–124, 2017.

[50] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.

[51] M. Imani, M. S. Razlighi, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Deep learning acceleration with neuron-to-memory transformation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 1–14.

[52] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, "Dual: Acceleration of clustering algorithms using digital-based processing in-memory," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 356–371.