Tri-HD: Energy-efficient On-chip Learning with In-memory Hyperdimensional Computing

Weihong Xu*, Saransh Gupta*, Justin Morris, Xincheng Shen, Mohsen Imani, Baris Aksanli *Member, IEEE*, and Tajana Rosing, *Fellow, IEEE*

Abstract—**The Internet of Things (IoT) has led to the emergence of big data. Processing this data, specially in learning algorithms, poses a challenge for current embedded computing systems. Brain-inspired hyperdimensional (HD) computing reduces several complex learning operations to simpler bitwise and arithmetic operations. However, it requires the use of large dimensional vectors, *hypervectors*, further increasing the amount of data to be processed. Processing in-memory (PIM) enables in-place computation which reduces data movement, a major latency bottleneck in conventional systems. In this paper, we propose Tri-HD, an in-memory HD computing architecture that performs HD classification in memory. To the best of authors' knowledge, Tri-HD is the first ReRAM PIM architecture to implement the complete HD computing-based classification pipeline including encoding, training, re-training, and inference for non-binary data. We also propose a novel distance metric that is PIM-friendly and provides similar application accuracy as the more complex baseline metric. Our proposed architecture is enabled in PIM by fast and energy-efficient in-memory logic operations. We exploit the voltage threshold-based memristors to enable single cycle operations. We also increase the amount of in-memory parallelism in our design by segmenting bitlines using switches. Our evaluation shows that for all applications tested using HD, Tri-HD provides on average 434× (2170×) speedup and consumes 4114× (26019×) less energy as compared to the CPU while running end-to-end HD training (inference). Tri-HD also achieves at least 2.2% higher classification accuracy than the existing PIM-based HD designs.**

Index Terms—**Processing in-memory, non-volatile memories, hyperdimensional computing, machine learning, classification, memristors, RRAM**

I. Introduction

The recent surge in interconnected devices and intelligent systems has led to the emergence of the big data phenomenon. In general, sophisticated learning methodologies are required to extract significant insights from this vast amount of data, entailing intricate computational processes. Consequently, these processes are typically executed on local systems equipped with multiple processor cores or transferred to the cloud for processing on large server infrastructures [1], [2]. On the contrary, the brain-inspired hyperdimensional (HD) computing paradigm reduces the complexity of operations by representing data in high-dimension space. Previous works have widely applied HD computing to various tasks such as image classification, language recognition, activity recognition, and bio applications [3]–[8]. These works have demonstrated the advantages of HD in terms of error reslience [3], [7] and energy efficiency [4], [5].

Even though HD computing is a type of efficient and general processing paradigm that exposes high data parallelism to low-level hardware, it still suffers from large energy and latency overheads when run on conventional von Neumann architecture. As discussed in previous studies [9], [10], the issue arises from the intensive data transfer between processing units and memory, attributed to the constraints of cache capacity and on-chip bandwidth. Processing in-memory (PIM) aims to mitigate this challenge by enabling the processing of a portion of the data directly within the memory, thus reducing the necessity to move all the data between the processing units and memory. Research interest in PIM has surged recently, largely due to the development of advanced non-volatile memory (NVM). Among the emerging NVMs, the memristor is regarded as one of the processing solutions to resolve the memory wall issue. Their rapid switch-over rate, minimal energy consumption during switching, and considerable scalability render them ideal for creating dense and swift PIM applications [11]–[14].

Several recent studies have utilized memristors to enhance PIM capabilities, as evidenced in [15]–[19]. There are two main streams to enhance the PIM capabilities for memristors. A type of common approaches involve modifying the sense amplifiers to support wider ranges of computational functionalities or to improve computing efficiency [16], [18], [20]. This method requires reading data from the memory, processing it using transistor-based circuits, and then storing the processed data back into the memory. However, a notable limitation of these designs is that the achievable data parallelism is restricted by the available quantity and capacity of the peripheral sensing circuitry around the memory. The other main stream exploits the bipolar switching behavior of memristors to implement logic in-memory [15], [17], [21]–[23]. Some of them implement logic purely in memory such as stateful implication logic [22], [24], and Memristor Aided loGIC (MAGIC) [21]. These designs apply various voltage levels to memory cells to represent multi-bit data while requiring no change in the peripheral circuits. They are purely in-memory operations which do not need to read out data but are restricted by the limited functionality they can implement. For example, MAGIC [21] only supports NOR directly in crossbar memory. All other functions are implemented by repeating multiple NOR cycles, which incurs significant runtime overhead.

Logic execution with MAGIC is fully compatible with the usual crossbar design, requires a lower number of voltages, and supports NOR which can be used to implement any Boolean

* Authors contribute equally.
S. Gupta is with IBM Research.
W. Xu, X. Shen, and T. Rosing are with the Department of Computer Science and Engineering, University of California San Diego, La Jolla, CA, 92093 USA (email: {wexu, xis143, tajana}@ucsd.edu).
J. Morris and B. Aksanli are with the Department of Electrical and Computer Engineering at San Diego State University, San Diego, CA, 92182 USA. J. Morris is also with the Department of Electrical and Computer Engineering at University of California San Diego, La Jolla, CA, 92093 USA. (email: justinmorris@ucsd.edu, baksanli@sdsu.edu).
M. Imani is with the Department of Computer Science, University of California Irvine, CA, 92697 USA (email: m.imani@uci.edu).

logic. Unlike implication logic based designs like IMPLY [22] which destroy one of the inputs, it is non-destructive. These properties of MAGIC make it a preferred logic family for resistive crossbar memories. MAGIC uses voltage threshold based memristors which switch whenever the voltage difference between the two terminals of the memory device exceeds a threshold. However, they don't fully utilize the threshold based switching of memristors and only implement NOR in crossbar memory. All other functions are derived using NOR, which results in unnecessary latency overheads. In our previous paper, FELIX [25], we proposed a purely in-memory implementation of fast and energy efficient logic. It extended the functionality of in-memory operations by implementing *single cycle* NOR, NOT, NAND, minority (Min), and OR directly in crossbar memory. We used these low-latency functions to implement functions like XOR and addition 2× faster than MAGIC [21]. Our design further increased the amount of in-memory parallelism by using in-block switches.

In this paper, we exploit single cycle operations to present a detailed PIM implementation of hyperdimensional (HD) computing, called Tri-HD. We, for the first time, design a ReRAM PIM architecture that can accelerate all the phases of HD pipeline namely, encoding, training, retraining, and inference. We present a new approximate distance metric, which is PIM-compatible unlike the traditionally used metrics. While this metric enables us to complete the HD pipeline in PIM, it comes at no loss in accuracy. We also discuss our proposed low latency functions from the perspective of vector operations. Apart from the typical data movement reductions that PIM designs achieve, we show how our PIM functions can provide large vector-wide parallelism. Moreover, in contrast to GPUs which are limited to some 1000s of cores, our PIM designs can offer much greater compute capability by making every memory array in the PIM chip a computing core.

We demonstrate the efficiency of our design, Tri-HD, by designing a HD computing accelerator with RRAM-based memory blocks. Our evaluation shows that for all applications tested using HD, Tri-HD provides on average 434× (2170×) speedup and consumes 4114× (26019×) less energy as compared to the CPU while running end-to-end HD training (inference). Tri-HD HD also achieves at least 2.2% higher classification accuracy than the existing PIM-based HD designs.

II. Background and Motivations

A. Basics of Hyperdimensional (HD) Computing

Brain-inspired Hyperdimensional (HD) computing is a computing paradigm which works based on understanding the fact that brains compute with *patterns of neural activity* [26]–[28], where such neural activity patterns can only be modeled with points of high-dimensional space (e.g., D=10,000). Classification is one of the most important supervised learning algorithms. Fig. 1-(a) shows the overview of HD computing architecture for a classification problem consisting of an encoder module and an associative memory. The goal of the encoder is to map an input data to a single hypervector with D dimensions and then combine these hypervectors for all of the images in a class to generate a unique hypervector representing each class. Each class hypervector is a long vector with D dimension, where each dimension can have binary values $\{0, 1\}$. Associative memory stores the trained hypervectors for all classes. In test mode, HD classifies an unknown input by encoding the input image to a hypervector using the same encoder used for training. The query hypervector has binary elements and the same D dimension as the class hypervectors. Next, associative memory checks the similarity of the query hypervector to all classes and classifies it to a class which has the closest similarity.

1) HD Encoding for Feature Vector

Fig. 1-(b) depicts the encoding module utilized in HD computing. Consider that every data point in the original space can be denoted by a features vector $\{v_1, \dots, v_n\}$. The objective of the encoding module is to transform this feature vector into a high-dimensional space, while preserving all the information within a high-dimensional vector. Each feature vector contains two types of information: the signal value and the index of each feature.

Feature values: To consider the impact of feature values, our design first identifies the minimum (v_{min}) and maximum value v_{max} that signal can take in all dimensions. Then, it quantizes the feature values into Q levels were v_{min} and v_{max} are the first and last levels respectively. HD assigns a single hypervector with D dimension to each of the quantized levels $\mathbf{L} = \{L_1, L_2, \dots, L_Q\}$ where $L_i \in \{0,1\}^D$ and L_1 and L_Q correspond to the v_{min} and v_{max}, respectively. The generation of the level hypervectors is similar to work [27], where the level hypervectors have similar values if the corresponding original data are closer, while L_1 and L_Q will be nearly orthogonal.

Feature index: To specify the impact of each feature index on encoded hypervector, HD generates a set of random identification hypervector $\mathbf{ID} = \{ID_1, \dots, ID_n\}$, where $ID_i \in \{0,1\}^D$ represents a hypervector corresponding to i^{th} feature index. Due to random generation, the ID hypervectors are semi-orthogonal, meaning that:

$$\delta(ID_i,\ ID_j) \simeq D/2, \quad \text{for } i \neq j. \tag{1}$$

Depending on feature values, each feature maps to one of the Q generated hypervectors. Hypervectors are combined together using element-wise XOR of the level and ID hypervector, and then summing the resulting hypervectors over all features:

$$H = \overline{L}_{v_1} \oplus ID_1 \ + \ \overline{L}_{v_2} \oplus ID_2 \ + \cdots + \ \overline{L}_{v_n} \oplus ID_n, \tag{2}$$

where $\overline{L}_{v_i}$ and ID_i denote the (binary) hypervector and feature index corresponding to the i-th feature of vector v.

2) HD Training and Retraining

The simplicity of HD training makes it distinguished from conventional learning algorithms. Consider hypervector H_i as the encoded hypervector of input i with the procedure explained above. Each input i belongs to a class j, so we further annotate H_i^j to show the class j of input i, as well. HD training simply adds all hypervectors of the same class to generate the final model hypervector. Therefore, the class hypervector of label j, denoted by C^j, is:

$$C^j = H_0^j + H_1^j + \cdots = \sum_i H_i^j \tag{3}$$

Meaning that we simply accumulate the encoded hypervectors for which their original input belongs to class j.

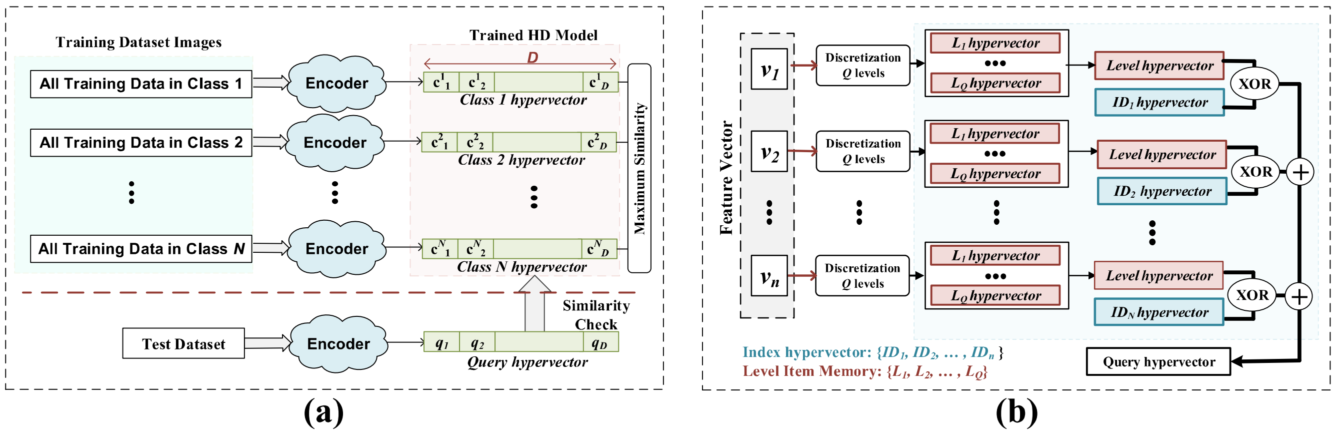


Fig. 1: (a) The overview of HD computing architecture for classification task. (b) The encoding module of HD computing mapping a feature vector with n elements to high dimensional space using pre-generated identity and level hypervectors.

Another advantage of HD over DNNs is HD supports efficient one-pass training, i.e., visiting each input just once and adding the H_is to create the model yields acceptable accuracy, while DNN training requires hundreds of iterations over the whole data set to converge to the final accuracy. HD accuracy can also be improved by *retraining* the model. During retraining, the encoded hypervector of each input is created again, and its similarity with the existing class (model) hypervectors is checked. If a *misprediction* is observed, say that encoded H^j belonging to class C^j is predicted as class C^k, the model is updated as follows, which means the information of H^j causing (mis)-similarity to C^k is discarded. The parameter α is the learning rate of the model.

$$\begin{cases} C^j = C^j + H^j\alpha \\ C^k = C^k - H^j\alpha \end{cases} \quad (4)$$

3) HD Inference

The inference step as well as the retraining step need to find out the most similar class hypervector to the encoded one. Most commonly, this is performed by cosine similarity while other metrics (e.g. Hamming distance) could be appropriate depending on the problem.

$$\arg\max_j \, cos(\vec{H},\vec{C^j}) = \arg\max_j \frac{\vec{H}\cdot\vec{C^j}}{||\vec{H}||\cdot||\vec{C^j}||} \quad (5)$$

Eq. (5) shows the similarity checking of encoded hypervector H with class hypervector C^j. Since classes are constant, $||\vec{C^j}||$ can be pre-calculated. $||\vec{H}||$ can be factored out as it is common for all candidate classes to be compared with H. Hence, cosine similarity reduces to a simple dot-product between H and C^js. These vectors are *not* in binary, they are the results of accumulating several other binary vectors.

B. Related Work and Challenges

1) Existing PIM-based HD Computing Work

Recent work has proposed ways to use PIM to accelerate HD computing. The associative memory data structure of HD computing is generally regarded as the most suitable candidate for acceleration by PIM. The work in [29] was the first to recognize this, and proposed an HD computing accelerator based on resistive CAMs. The design achieved 746× reduction in energy-delay product (EDP) as compared to the CMOS-based ASIC proposed in [3]. The authors in [30] tried to alleviate the thermal challenges related to PIM implementation of associative memory. They proposed memory block selection and activation schemes to reduce the overall chip temperature, resulting 57.2% memory lifetime improvement and 17.6% performance gain.

The design in [31] extended the idea of the work in [29] and supplemented it with a digital HD mapper and encoder to accelerate complete HD algorithm. However, the encoding schemes used by them do not provide state-of-the-art results. The work in [32] presented an HD-chip which implemented both encoding and search operations using a combination of carbon nanotube field-effect transistors (CNFETs) and ReRAM cells to achieve low EDP. However, they implement only inference and supported only few specialized applications. The work in [33] proposed crossbar memory based encoding and associative search. The encoding module used analog computing primitives to perform bitwise logical AND of hypervector dimensions. The demonstrated chip provided 6.6× energy and 3.8× area improvement. However, approximately 92% of the area and 89% of energy were consumed by memory peripherals.

Another set of designs in [34], [35] take a different approach where instead of implementing associative memory using PIM, they used large ReRAM PIM-based XOR operations to compare different hypervectors. In both the implementations, HD encoder used ReRAM PIM bitwise operations to generate hypervectors. A similar work in [36] breaks down all HD operations into dot product and bitwise operations and implements them using analog computing techniques. Finally, recent research in [37] presented a PIM implementation of both HD training and inference. To maintain high training accuracy while achieving the efficient bitwise computation, the work proposed the use of stochastic training, which generated multiple binary hypervectors for each class. It increased the baseline accuracy, which was still 4-9% less than what HD could achieve.

2) *Challenges of Existing Work*

Most of prior work tried to accelerate HD by speeding up the computation in associative memory [29]. In this work, we use HD computing for practical classification problems such as speech recognition [38], face recognition [39], activity recognition [40], and physical monitoring [41]. We observe that in most tested applications, the data point in original data is a long feature vector. Encoding such feature vector to high dimensional space is extremely costly.

TABLE I: Energy consumption of HD encoding module and associative memory for different applications

	Encoding Module	*Associative Memory*
Speech Recognition	8.18 *mJ*	8.78 *mJ*
Face Recognition	7.85 *mJ*	1.43 *mJ*
Activity Recognition	7.01 *mJ*	3.87 *mJ*
Physical Monitoring	0.23 *mJ*	2.25 *mJ*

Table I compares the energy consumption and execution time of HD computing for several applications including language recognition, speech recognition, face recognition, and activity recognition. The experiments have been performed on the Intel i7 CPU with 16GB memory. Our evaluation shows that for practical problems, the encoding module is a dominant part of energy consumption and execution time. For example, for four tested applications the encoding module takes around 60% of total energy.

III. PIM Logical Implementation in Tri-HD

In this section, we present how Tri-HD utilizes the ReRAM crossbar array to realize in-memory Boolean and logical computations.

A. *Overview*

The HD algorithm in Section II is lightweight but memory intensive because it only requires simple XOR and addition operations. The bottleneck is the expensive data movement to fetch hypervector from the memory. The majority of the energy in Table I is consumed by data movement instead of arithmetic operations. To address these challenges, Tri-HD leverages the PIM techniques that have been demonstrated energy-efficient for memory-intensive workloads [18], [25], [42], [43]. The PIM-based computation reduces most of the data movement overhead during computation, which improves the overall efficiency.

Tri-HD computes vector-wide logical operations, meaning that the entire hypervector is processed in parallel across the crossbar row or column. Specifically, Tri-HD employs a variable voltage-based execution scheme, where the operation to be performed is determined by the applied voltage. Furthermore, instead of solely relying on the resetting behavior of memristors, we leverage their two-way switching to enhance the capabilities of PIM. When the voltage $V_{pn} > v_{on}$, a memristor transitions from a high resistive state (R_{OFF}) to a low resistive state (R_{ON}). Conversely, when $V_{np} > v_{off}$, it switches from R_{ON} to R_{OFF}. Here, v_{on} and v_{off} represent the device-specific voltage thresholds, while V_{pn} denotes the voltage difference between terminals p and n. Tri-HD extensively supports various types of logical operations. These operations are divided into two categories: single-cycle and multi-cycle operations, based on the number of computing cycles needed. The single-cycle

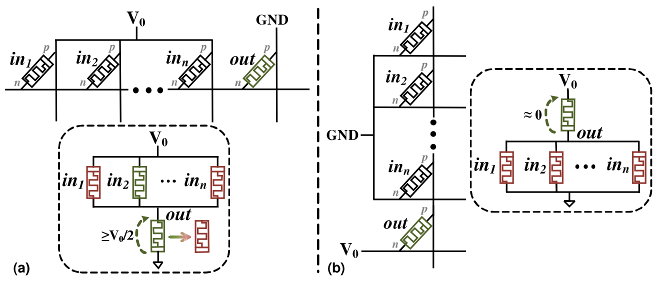


Fig. 2: n-input NOR implementation in (a) a row and (b) a column.

operations include: NOR, OR, NAND, and Min. The multi-cycle operations include: Maj, AND, XOR, and addition.

B. *Single-Cycle Operations*

NOR: Fig. 2 illustrates the implementation of NOR in a memristor crossbar array. Initially, the output memristor is set to R_{ON}. To perform NOR operation in a row, an execution voltage V_0 is applied to the p terminals of the inputs, and the p terminal of the output memristor is grounded as in Fig. 2-(a). On the other hand, when NOR is executed in a column, the n terminals of the inputs are grounded, while V_0 is applied to the n terminal of the output, as shown in Fig. 2-(b). The purpose of both executions is to switch the output memristor from R_{ON} to R_{OFF} whenever the NOR output is '0'.

Assume two vectors A and B with 100 1-bit elements each are stored in the memory such that ith element of a vector is present in the ith row of the memory. Moreover, all elements of the vector A (B) occupy the ath (bth) column. We call this way of storing vectors column-wise storage. To perform a NOR over the ith element of the two vectors, we implement the 2-input row implementation of NOR. As discussed before, we apply voltage V_0 to the bitlines a and b, while grounding the output bitline, say c. However, we notice that this voltage application remains the same regardless of the value of i. Hence, the NOR operation discussed above provides vector-wide parallelism, where an operation over all the elements of a vector takes the same time as the operation over a single element of the vector. This discussion can be similarly extended to the case where the vectors are stored row-wise and we use NOR in a column implementation.

NAND and Min: Tri-HD arithmetic operations do not depend on the specific inputs, but rather on the voltage across the n and p terminals of the output device. This allows to implement minority and NAND operations in memory. For example, let's consider a 3-input NOR gate with a voltage of $V_0 = 1$V and an offset voltage of $v_{off} = 0.5$V. When the inputs are '000,' '001,' '011,' and '111,' the voltage across the output memristor is 0V, 0.5V, 0.67V, and 0.75V, respectively. In this case, the output memristor switches in all cases except the first one. Now, if we change the value of V_0 to 0.75V, the voltage across the output memristor changes to 0V, 0.38V, 0.5V, and 0.56V for the same inputs. In this case, the output switches to R_{OFF} only when there are at least two '1's in the input. This configuration effectively implements the 3-bit minority function (Min3). Furthermore, if we decrease V_0 to 0.67V, the output changes only when the inputs are '111,' resulting in an output of '0' only when all the inputs are '1.' This operation is

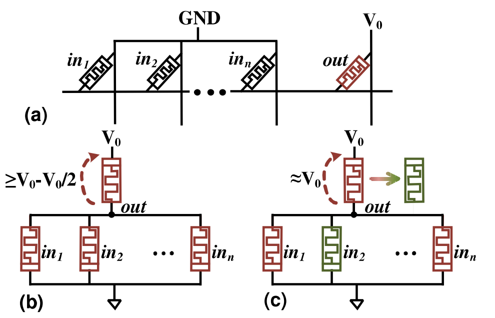


Fig. 3: Voltage division for Tri-HD OR. (a) Application of voltage for OR, (b) output memristor remains R_{OFF} when all the inputs are R_{OFF} (red), and (c) output memristor switches to R_{ON} when one or more inputs are R_{ON} (green).

equivalent to a 3-bit NAND operation. The above logic can be extended to implement N-bit minority and NAND functions. The execution voltage V_0 required to implement these functions can be calculated using Equation (1), where N is the number of inputs.

$$\left(\frac{v_{off}}{R_{ON}}\right)\cdot\left\{R_{ON}+\left(\frac{R_{OFF}}{N-(n+1)}\right)||\left(\frac{R_{ON}}{n+1}\right)\right\}<V_0<\left(\frac{v_{off}}{R_{ON}}\right)\cdot\left\{R_{ON}+\left(\frac{R_{OFF}}{N-n}\right)||\left(\frac{R_{ON}}{n}\right)\right\}, \tag{6a}$$

$$\left(\frac{n+2}{n+1}\right)\cdot v_{off}<V_0<\left(\frac{n+1}{n}\right)\cdot v_{off} \tag{6b}$$

The value of n is defined by the operation to be executed. For Min, $n=\lceil N/2\rceil$ and for NAND, $n=N$. Eq. (6b) is an approximation of Eq. (6a) under the assumption that $R_{OFF}>>R_{ON}$.

Therefore, apart from NOR and NOT, the Tri-HD also supports a single cycle MinN and NAND. In theory, this technique can be expanded to any value of n. However, the practical feasibility of implementing large values of n is challenged by the unavailability of different voltage levels. For instance, to implement a 6-bit NAND, Tri-HD requires a V_0 of 0.58V. This value changes to 0.57V and 0.56V for a 7-bit and 8-bit NAND, respectively. Generating these distinct and closely valued voltage levels reliably is a difficult task. Therefore, to ensure practicality, we limit the Tri-HD to 2-bit and 3-bit NAND and Min operations. The vector-wide parallelism provided by these and future operations can be inferred from the discussion on NOR operations.

OR: Tri-HD exploits the behavior of the memristor device to reduce the latency of the OR operation to one cycle. As explained in Section I, a device can be switched from R_{OFF} to R_{ON} by applying a voltage greater than the threshold, v_{on}. In contrast, when using MAGIC to implement OR in crossbar memory, it requires two NOR cycles since it relies solely on the resetting behavior of memristors. The voltage division for different possible inputs is shown in Fig. 3. The ground and V_0 terminals are used in the opposite manner compared to MinN. When all the input and output memristors are R_{OFF}, the voltage across the output memristor is significantly lower than V_0. However, if one or more inputs are R_{ON}, the voltage across the output is approximately V_0. If V_0 exceeds v_{on}, then the output memristor switches to R_{ON}.

The aforementioned behavior is utilized to implement the OR operation in memristive memory. Initially, the output memristor

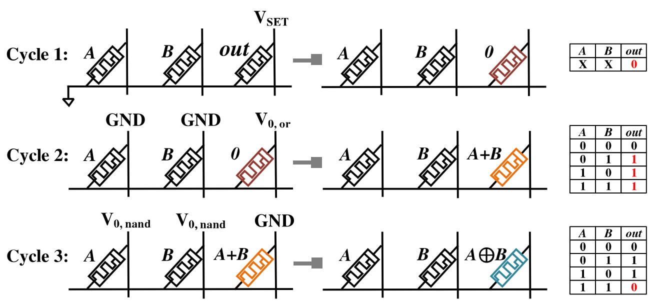


Fig. 4: Different stages in implementing 2-bit XOR using Tri-HD

is set to R_{OFF}. To perform the OR operation in a row, the p terminals of the input memristors are connected to ground while V_0 is applied to the p terminal of the output. In the case of OR operation in a column, V_0 is applied to the n terminals of the inputs and the n terminal of the output is connected to ground (refer to the figure illustrating the p and n terminals). If the states R_{ON} and R_{OFF} of the memristor represent the logical 'high' and 'low' states, respectively, the result of the OR operation corresponds to R_{OFF} when all the input bits are low, and R_{ON} otherwise. The execution voltage, V_0, required to implement OR is given by,

$$\frac{|v_{on}|}{R_{OFF}}\cdot\left\{R_{OFF}+(R_{ON})||\left(\frac{R_{OFF}}{N-1}\right)\right\}<V_0<\frac{|v_{on}|}{R_{OFF}}\cdot\left\{R_{OFF}+\left(\frac{R_{OFF}}{N}\right)\right\}, \tag{7a}$$

$$\left(1+\frac{R_{ON}}{R_{OFF}}\right)\cdot|v_{on}|<V_0<\left(\frac{N+1}{N}\right)\cdot|v_{on}|, \tag{7b}$$

where N is the number of inputs. Eq. (7b) is an approximation of Eq. (7a) under the assumption that $R_{OFF}>>R_{ON}$.

C. Multi-Cycle Operations

The in-memory operations proposed in the above section can be combined to extend the functionality of the memory to realize multi-cycle operations.

Maj and AND: Majority (MajN) and AND can be implemented by inverting MinN and NAND respectively. This results in 2-cycle MajN and AND in Tri-HD in contrast to four cycles in MAGIC.

XOR: XOR ($\oplus$) can be expressed in terms of OR ($+$), AND (.), and NAND ($(.)'$) as follows:

$$A\oplus B=(A+B).(A.B)' \tag{8}$$

Fig. 4 shows the in-memory implementation of Eq. (8). Instead of calculating OR and NAND separately and then ANDing them, we first calculate OR and then use its output cell to implement NAND. In this way, we eliminate separate execution of AND operation. This logic just requires 2 Tri-HD cycles and three memristor cells. Two of the cells work as the input cells to store operands A and B, respectively. The third cell works as the output cell. In contrast, the state-of-the-art PIM technique proposed in [17] uses 5 cycles and 5 memristors for the fastest XOR implementation, while the most area conservative approach takes 7 cycles and 3 memristors. Hence, the proposed XOR implementation is 1.4× faster and 1.7× smaller.

Addition: Tri-HD implements addition by combining XOR and MajN operations. A 1-bit adder can be represented by,

$$\begin{aligned}S&=A\oplus B\oplus C_{in},\\C_{out}&=A.B+B.C+C.A=MajN(A,B,C_{in}),\end{aligned} \tag{9}$$

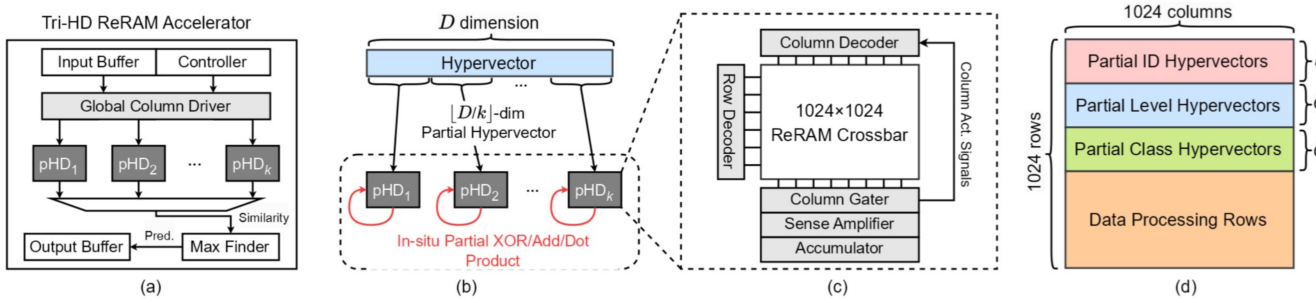


Fig. 5: Tri-HD design overview: (a) Overall architecture of ReRAM-based Tri-HD accelerator, (b) Distribution of partial hypervectors, (c) Internal architecture of pHD ReRAM crossbar, and (d) data organization in each ReRAM crossbar.

TABLE II: PIM performance improvements of Tri-HD over MAGIC [21].

Property	Design	NOR3	NAND3	Min3	OR3	Maj3	AND3	XOR2	1-bit ADD
Latency (Cycles)	MAGIC	1	5 (6)	5 (6)	2	4	4	5 (7)	12 (14)
	Tri-HD	1	1	1	1	2	2	2	6
	Improv.	1×	5 (6)×	5 (6)×	2×	2×	2×	2.5 (3.5)×	2 (2.33)×
Memory (# of Cells)	MAGIC	1	5 (4)	5 (4)	2	4	4	5 (3)	12 (6)
	Tri-HD	1	1	1	1	2	2	1	4
	Improv.	1×	5 (4)×	5 (4)×	2×	2×	2×	5 (3)×	3 (1.5)×
Energy (fJ)	MAGIC	24.11	120.17	120.38	48.12	96.17	96.15	120.29	288.82
	Tri-HD	24.11	49.24	41.64	9.53	65.65	73.26	34.97	135.60
	Improv.	1×	2.44×	2.89×	5.05×	1.47×	1.31×	3.44×	2.13×

where A, B, and C_{in} are 1-bit inputs while S and C_{out} are the generated sum and carry bits respectively. Here, S is implemented as two serial in-memory XOR operations. C_{out}, on the other hand, can be executed by inverting the output of MinN. Hence, S takes a total of 4 cycles and 2 additional memristors, while C_{out} needs 2 cycles and 2 additional memristors.

The in-memory processing techniques previously proposed also support addition within the crossbar memory [17]. These approaches break down an operation into a series of NOR operations. A typical addition implementation requires 12 NOR operations, resulting in 12 MAGIC NOR cycles [17] as compared to 6 in Tri-HD and 12 additional memristors compared to 4 in Tri-HD.

D. Performance Comparison

Table II compares the execution of different Boolean logic functions in Tri-HD with previously proposed PIM techniques. The latencies in the table and the discussion exclude the first initialization cycle which is common to all designs. The numbers in brackets represent the properties of the area conservative designs [17]. It shows that Tri-HD performs the same as or significantly better than the fastest state-of-the-art technique [17], [21]. For example, for addition, Tri-HD is 2× faster, has 2× better energy efficiency, and 3× lower memory size. In the following subsections, we outline the implementation of basic boolean operations in Tri-HD and how they can be used to perform vector-wide operations that form the basis of Tri-HD architecture.

IV. Tri-HD Acceleration in ReRAM

In this section, we present the ReRAM-based PIM accelerator, Tri-HD, to accelerate HD computing pipeline. Tri-HD leverages the PIM logical operations introduced in Section III to avoid data movement due to the memory-intensive HD operations.

A. Overview

Fig. 5-(a) shows the overall architecture of Tri-HD accelerator based on ReRAM. Tri-HD is composed of input/output buffers, k partial HD (pHD) modules, global column driver, and a max finder. The acceleration of HD encoding, inference, and training/retraining is achieved by utilizing the PIM logical operations in Section III. The storage and computation of hypervectors are performed in the pHD module, thus avoiding the overhead of hypervector data movement. As shown in Fig. 5-(c), each pHD module consists of a 1024×1024 ReRAM crossbar, row/column decoders used to activate wordline (WL) and bitline (BL). The sense amplifier converts data from ReRAM cells to digital signals that can be processed by the accumulator. The column gater is used to deactivate unused rows and cooperate with the approximate similarity search as explained in Section V-A.

B. Inter-crossbar Parallelism with Partial Hypervector

The HD algorithm requires a high dimension D (2,000 to 10,000) to achieve satisfactory classification accuracy. This dimension is much larger than the size of the crossbar array (1024) used in this work, which means that a single BL is unable to accommodate the D-dimensional hypervector. One design choice is to segment hypervectors in each crossbar. However, this requires sequential processing for each segment of the hypervector, increasing the overall latency.

We observe that the HD algorithm in Section II-A exposes high data parallelism to hardware: XOR and addition operations over D dimensions during encoding, inference, and training/retraining can be performed in parallel. In Fig. 5-(b), Tri-HD achieves inter-crossbar parallelism by uniformly partitioning each D-dimensional hypervector into k smaller partitions, called partial hypervectors. In this case, each pHD module receives one $\lceil D/k \rceil$-dimensional partial hypervector. All partial hypervectors in different crossbar arrays can be calculated in parallel.

Fig. 5-(d) illustrates the hypervector and data organization in each crossbar array. Assume a HD algorithm with n ID, Q level, and C class hypervectors, all partial hypervectors (ID, level, and class) are horizontally organized on rows, consuming a total of $n+Q+C$ rows. Meanwhile, the rest of the rows are reserved for storing the intermediate data. This data organization is compatible with the requirements of PIM logical operations. The row address space is managed by the controller, where associated addresses are loaded into the row decoder to perform different arithmetic functions.

C. Tri-HD Encoding

For a HD configuration composed of n elements and Q levels hypervectors, we have $n+Q$ hypervectors and each one has D

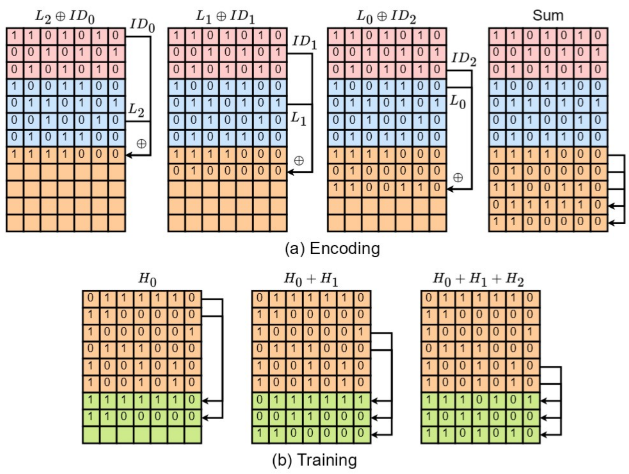


Fig. 6: Illustration of in-memory Tri-HD encoding and training with $n = 3$ and $Q = 4$.

dimension. All these vectors are generated once and pre-stored in $n+Q$ rows of the crossbar subarray. Fig. 6-(a) shows an example of HD encoding using the PIM-enabled crossbar in Tri-HD. Here, the feature vector is $[2,1,0]$, where $n = 3$ and $Q = 4$. The encoding consists of $2n-2$ steps, where the first n steps perform n PIM XORs while the remaining $n-2$ steps sum up the XORed results. During the XOR step, each ID hypervector ID_i is XORed with one of the level hypervector L_j based on the feature values. For a pair of ID and L, Tri-HD XOR can be computed in parallel for all dimensions since all dimensions of a vector are stored in the same row. Each XOR operation requires one additional memory row to store the output.

The results of the XOR operations are then summed up dimension-wise. Tri-HD counts the number of 1s in all XOR results for each dimension by executing the PIM addition operation. In order to perform addition for all the dimensions in parallel, we store the output of addition vertically in a column, instead of a row, as shown in Fig. 6-(a). We add n elements serially, three bits at a time. If $X_1,\ X_2,\dots\ X_n$ are the vectors to be added together, we first add $X_1,\ X_2$, and X_3 to generate S_1 and C_1. We then add $X_4,\ X_5$, and $\{C_1,\ S_1\}$ to generate S_2 and C_2, and so on till we have added all XOR results. The addition of n 1-bit elements results in an output with $p = \lceil log_2 n \rceil$ bits, requiring p rows to store the output of addition. In addition, Tri-HD also requires $p+1$ rows to store the intermediate results of addition like C_1, S_1, S_2, C_2, etc. Instead of computing all XOR results and then summing, we limit the addition of XOR results to a maximum of three at a time. Therefore, three XORed results are computed in pairs,where we compute three and add them together. This approach reduces the memory needed for XOR results from n rows to only 3. In addition to the 3 rows for XOR results and p rows for the addition results, $p+1$ rows are needed to store the intermediate results.

D. Tri-HD Training

The HD training in Tri-HD involves class-wise addition of the hypervectors generated by encoding. This addition is similar to the addition in encoding. The difference is: instead of summing up 1-bit hypervectors, the training process adds multiple integer or binary hypervectors. HD training is highly parallel and allows us to split training into multiple modules, where each module *independently* performs (partial) training over a subset of the data and then the partially trained class hypervectors are added together. To implement this, we fuse the HD encoding and training for partial hypervectors in each pHD module. For each input feature vector, the encoding is first performed as explained in Section IV-C. Then, instead of generating a new encoded hypervector for each input, we keep on accumulating the encoding outputs to a single hypervector as in Fig. 6-(b). Hence, at any point during the computation, only one partially trained class hypervector is stored in each pHD module. This scheme saves the required memory space during training.

E. Tri-HD Inference

Tri-HD inference involves 2 steps: 1. Tri-HD encoding and dot product-based similarity search. First, the encoding step converts the input feature into the corresponding hypervector. The encoded vectors are then multiplied with class hypervectors to find the best match. In Section V-A, we present an efficient approximation to implement the similarity search. The output of the similarity search is the closest class, and the corresponding label is regarded as the prediction label.

F. Tri-HD Re-training

The re-training is the combination of inference and training. First, the input feature is encoded and passing through inference step. The prediction output of the inference is compared to the true label. If the inference output matches the label, we bypass the remaining stages of retraining. Otherwise, the encoded hypervector is used to update the corresponding class hypervector. In addition, the hypervector is also used to update the inferred class. However, for the inferred class, *pHD* performs hypervector subtraction instead of addition. This process is carried out for a maximum number of iterations provided by the application parameters, and the model corresponding to the best accuracy is chosen as the final trained model.

V. Tri-HD Optimization

In this section, we present several optimization schemes to reduce energy consumption while improving memory utilization for Tri-HD accelerator.

A. Efficient Approximate Similarity Search

A key operation of HD pipeline is the dot product-based similarity search, which is used during retraining and inference. The dot product of two vectors can be broken down into their element-wise multiplication and the accumulation of the generated product vector. To implement element-wise multiplication, the two hypervectors are stored row-wise so that a column of the memory contains the element from each vector with the same index. Then, row-parallel in-memory multiplication is executed to obtain the element-wise product using the switching techniques in [23], [25]. However, this process is expensive because the vector accumulation involves adding D elements together, which is a slow and serial operation due to the large dimensionality.

To improve the speed and efficiency of the similarity search, we take advantage of the error resilience of HD algorithm [6],

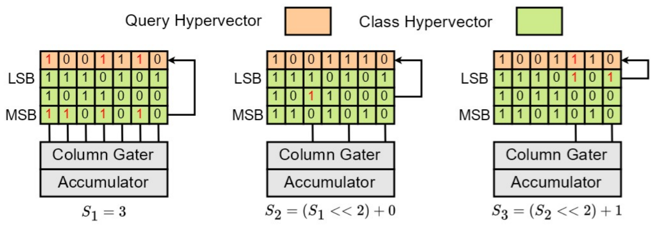


Fig. 7: Approximate search with power-of-2 search in Tri-HD.

[44] to compute the approximate version of exact dot product at the cost of negligible accuracy loss. Tri-HD first quantizes all elements in hypervector to the maximum power of 2 that is less than each element's value, which can be expressed as:

$$\text{Quantize}(x) = \lfloor \log_2 x \rfloor. \quad (10)$$

There are two possible ways of applying this approximation: before or after the dot product accumulation. We call them the *pre-DP approximation* and *post-DP approximation*, respectively. The post-DP approximation, in theory, leads to the least quality loss. However, this would still require fixed-point multiplication over thousands of dimensions, which is still slow and energy-consuming. Hence, we evaluate and compare the accuracy loss caused by the pre-DP and post-DP approximations in Fig. 11. It shows that both two approximation schemes are robust to computational errors, resulting acceptable accuracy loss compared to the exact dot product computation. This robustness to noise scales with dimensionality: the accuracy gap between approximation and baseline reduces as D increases. Therefore, Tri-HD uses a dimensionality of $D = 10{,}000$ to achieve the highest robustness and minimal accuracy loss.

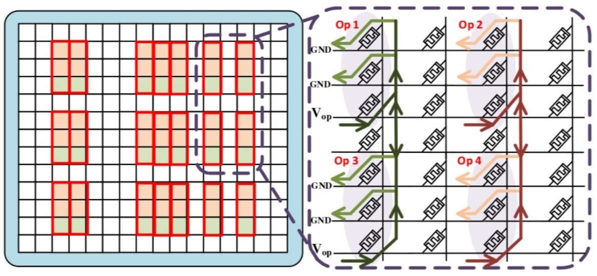
Both pre-DP and post-DP approximations described above involve quantizing the hypervector to the maximum power of 2 less than the original value (see Eq. (10)). In hardware, this is equivalent to finding the leading (trailing) one for positive (negative) numbers [45]. We utilize exact search operations to implement this power-of-2 (P-of-2) search shown in Fig. 7. In this example, the quantized class hypervector (C*) is used to shift the query hypervector. The shifting is equivalent to multiplication of input with C*. The output of this shift is again quantized with P-of-2 to perform the final accumulation. The P-of-2 search starts from the most significant bit (MSB) to the least significant bit (LSB). Meanwhile, it is performed in a row-parallel manner. For each column, whenever a '1' is detected, the column gater controls corresponding rows to be deactivated from further search operations. We implement a counter in the column gater which counts the number of rows that are selected for each bit-column. At the end, the values in the counter are multiplied by the corresponding power of 2 and added to generate the approximation similarity.

We evaluate the performance and energy consumption for the pre-DP and post-DP approximation in Section VI-E. The results show that the pre-DP approximation is 3.4× faster and 2.3× more energy efficient compared to the post-DP approximation while providing on average 0.16% better accuracy. In this work, we mainly focus on the pre-DP approximation scheme.

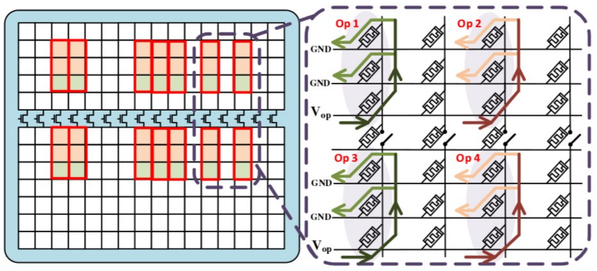
B. Intra-crossbar Parallelism

To enhance the level of parallelism within the pHD module, we divide the array into smaller partitions, where the degree of parallelism achievable is directly proportional to the number of partitions within the memory block. These partitions are created using isolation transistors that separate BLs into smaller segments. This allows Tri-HD to perform multiple operations independently at the same time. Consider a subarray with 64-bit WL and a capacity of 1024 words. We need to perform a bitwise OR operation between 10 pairs of words and store the output in the memory. All these steps are independent of each other and can occur simultaneously if the memory supports it. If the memory has 10 or more partitions, Tri-HD can execute this task in a single cycle. In comparison, the baseline requires 10 steps to complete the task, with each step implementing 64 parallel single-cycle OR operations between a pair of words [17].

Fig. 8-(a) illustrates the behavior of Tri-HD in a memory without partitioning when parallelizing operations across rows and columns concurrently. Currents from different operations interfere with each other, as depicted by {Op1 Op3} and {Op2 Op4} in Fig. 8-(a). Consequently, it effectively results in a single operation with increased inputs and multiple copies of output. To address this, transistors are used to divide the BLs while maintaining their logical equivalence, shown in Fig. 8-(b).



(a) Limitation of the memory in implementing operations in a row and column simultaneously. The crossbar structure does not distinguish the currents from different operations.



(b) Tri-HD with transistors and the resultant memory crossbar. The currents for the four operations do not interfere with each other.

Fig. 8: Intra-crossbar parallelism with isolation transistors

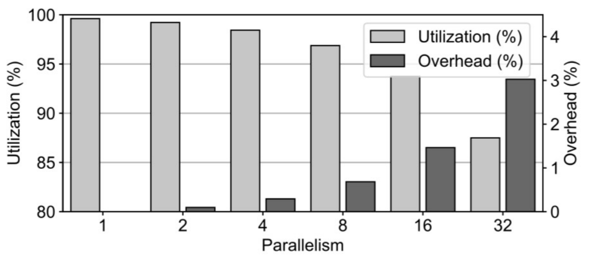


Fig. 9: The relationship between memory utilization and area overhead due to isolation transistors with increased parallelism in Tri-HD.

By switching off the transistors, the currents associated with different operations are prevented from merging. This capability allows Tri-HD to parallelize operations simultaneously. Ideally, our goal is to maximize the number of partitions. However, there are drawbacks associated with increasing the number of partitions. Firstly, a higher number of partitions results in reduced memory utilization. Since each partition requires its own processing elements, increasing the number of partitions also increases the number of devices needed. Consequently, when the memory size is fixed, increasing the number of processing elements directly decreases the amount of memory available for storage. Secondly, a larger number of partitions require a higher number of transistors to segment the bitlines, resulting in increased area overhead. Fig. 9 illustrates the variations in memory utilization and area overhead for a 1024×64 memory block as the number of partitions increases.

C. Endurance Management

A comparatively lower endurance of ReRAM cell is the main factor limiting the life of Tri-HD. This problem worsens when memristors are used for PIM as it requires frequent switches. Table II shows that Tri-HD's PIM operations reduce the number of switches compared to the previous work, helping to extend the lifetime of the memory. However, having such a memory is still less feasible without an endurance management policy.

The most frequently switching area is the data processing rows in Fig. 5-(d) because all operations use these rows to store and process intermediate data. This results in differential degradation where some part of the memory becomes unusable before the other. To alleviate this issue, Tri-HD adopts a simple and effective endurance management policy that ensures even memory degradation. Tri-HD changes the row allocation for data processing over time. In hardware, this is achieved by adding a simple linear feedback shift register (LFSR) [46] to the controller. The addresses sent to the row decoder will first pass through LFSR to distribute the accessed addresses. As a result, the endurance management policy distributes the degradation across different rows, effectively extending the lifetime of the device.

The adoption of endurance management policy can increase the device life time by 2.8× to 1.7× for tasks in Table III. We also estimate the lifetime of Tri-HD in terms of the number of classifications that can be performed. Each Tri-HD inference requires n XOR and n addition operations, where n denotes the number of ID hypervectors. Each inference requires $5n$ writing operations. Under the assumption of 10^9 ReRAM life time, the estimated number of classifications is 3.3×10^8 to 7.6×10^9. Moreover, we also observed that some ReRAM devices [47] provide even longer write endurance at the level of 10^{12}. The life time of Tri-HD accelerator can be further extended by using these devices.

VI. Evaluation

A. Experimental Setup

We compare Tri-HD performance and energy efficiency with Intel i7-8700K CPU @ 3.7GHz (12 cores) with 16GB memory and 256GB SSD. All software support for application level evaluation including training and testing of HD model have been performed in CPU using Python implementation. For hardware level evaluation, we have developed a cycle accurate simulator which emulates the behaviors of Tri-HD. Our simulator pre-stores the randomly generated level and index hypervectors in memory and performs the encoding, training, retraining, and inference operations in-memory using the controller signals.

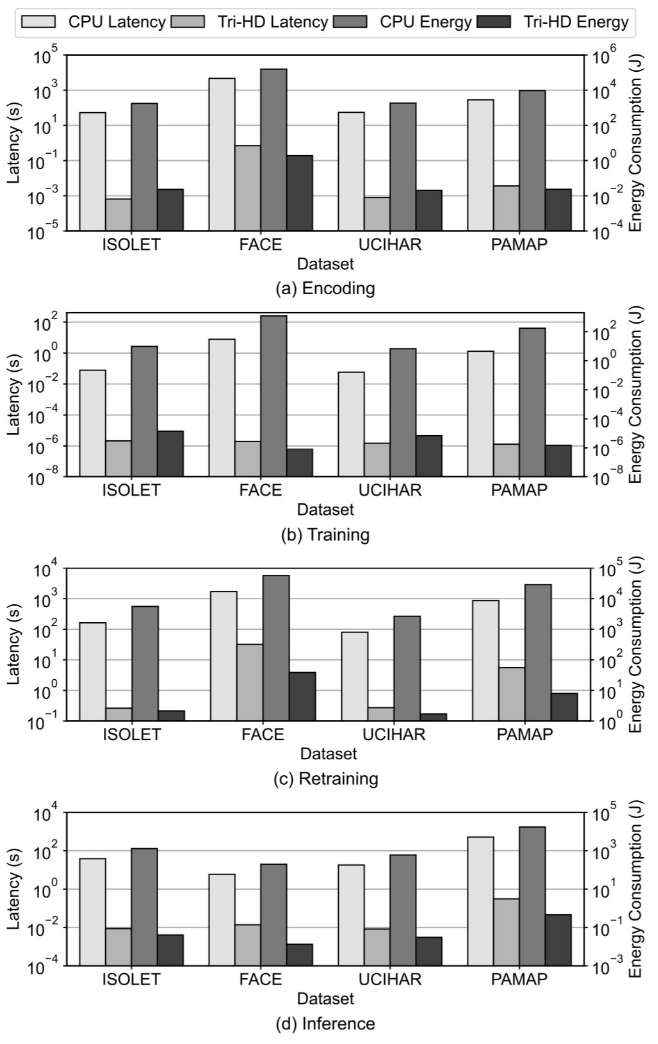


Fig. 10: Latency and energy consumption of HD on CPU and the proposed Tri-HD accelerator.

We extracted the circuit level characteristic of Tri-HD performing basic bitwise operations and give them as input to simulator. Performance and energy consumption of proposed hardware are obtained from circuit level simulations for a 45nm CMOS process using Cadence Virtuoso. We use VTEAM memristor model [48] for our memory design simulation with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively.

TABLE III: Specifications of evaluated datasets.

Dataset	Class	Feature	Train Size	Test Size
ISOLET [38]	26	617	6,238	1,559
FACE [39]	2	608	522,441	2,494
UCIHAR [40]	12	561	6,213	1,554
PAMAP [41]	5	27	122,218	101,582

B. Workloads

We consider the following four types of real-world datasets to evaluate the efficiency of proposed Tri-HD: 1. **ISOLET** [38] for speech recognition; 2. **FACE** [39] for face detection; 3. **UCIHAR** [40] for activity recognition; 4. **PAMAP** [41] for physical activity monitoring. The specifications for these datasets are summarized in Table III.

TABLE IV: The energy efficiency, speedup and memory efficiency of Tri-HD compared to MAGIC.

	ISOLET	**FACE**	**UCIHAR**	**PAMAP**
Energy Improv.	2.20×	2.20×	2.21×	2.26×
Speedup	1.86×	1.86×	1.88×	1.87×
Memory Efficiency	1.61×	1.61×	1.61×	1.82×

C. Tri-HD Results

We first compare the efficiency of HD computing with dimension $D = 10,000$ for three different platforms: CPU, proposed in-memory Tri-HD architecture, and MAGIC [17]. Table IV compares the energy efficiency, speedup and memory efficiency of Tri-HD with MAGIC [17] while running different HD classification applications. The memory efficiency is defined as the number of processing cells required to execute in-memory operations. For a fair comparison, we use the proposed architecture to evaluate both Tri-HD and MAGIC. The results show that Tri-HD HD can achieve on an average 2.21× higher energy efficiency, 1.86× speedup, and 1.68× lower memory requirement as compared to MAGIC. Moreover, both FELIX [25] and MAGIC [21] enable dot product with parallel multiplication followed by a series of addition operations. In contrast, Tri-HD uses an approximate version of dot product which does not cause any inference accuracy loss for HD applications. Tri-HD's dot product with rounding before multiplication is 858× faster and 1.8× more energy efficient than dot product in FELIX. This is a direct result of replacing multiplication and serial addition operations with faster parallel in memory search operations.

Fig. 10 shows the energy consumption and execution time of HD encoding, training, retraining, and inference for different application on CPU and Tri-HD. Our evaluation shows that for all the applications tested, Tri-HD provides on average 434× and 2170× speedup and 4114× and 26019× lower energy consumption as compared to the CPU while running end-to-end HD training and inference respectively. End-to-end HD training involves encoding and training, followed by 64 iterations of retraining to achieve maximum accuracy. The latency does not depend much on the number of classes in a dataset in Tri-HD. Tri-HD exploits the fact that computation for each class is independent and executes them in parallel. This is specifically evident in the case of ISOLET and FACE datasets, where Tri-HD latency of end-to-end training mainly depends upon the number of training samples. Moreover, the iterative nature of retraining makes it the slowest operation in the HD pipeline for both Tri-HD and CPU. But the extensive parallelism offered by Tri-HD makes in-memory retraining 276× faster than CPU. Here, we report the result for Tri-HD with single memory partition. The higher efficiency of the Tri-HD comes from: 1. memory compatible operations of HD which enables Tri-HD to parallelize the operations in different dimensions and, 2. lower data movement and higher locality of the data in-memory for Tri-HD computation. Applications with large number of features require more resources in order to perform the computation. Similarly, for each application Tri-HD efficiency can change depending on the number of partitions that each memory uses as shown in Section V-B.

D. Tri-HD vs Previous Work

Here, we compare Tri-HD with existing PIM-based design for HD computing. It should be noted that Tri-HD is the first works that uses 32-bit dimensions for hypervector, in contrast with just 1-bit dimensions used by all other approaches. This increase both the algorithmic and hardware complexity in Tri-HD. However, owing to the higher bitwidth, Tri-HD is able to achieve much higher classification accuracy as compared to the existing work. For example, the model obtained from Tri-HD is 2.2% more accurate (Fig. 11) as compared to RRAM-based design in [35]. At the same time, our end-to-end training with 20 retraining iterations is 48× faster and 3.4× more energy efficient. The speedup is a result of highly parallel and memory-compatible operations implemented in Tri-HD, whereas the approximate similarity metric of Tri-HD reduces the otherwise high energy consumed by more complex operations. While comparing both designs without retraining, Tri-HD is 289× faster and 5.2× more energy efficient.

Other PIM accelerators for HD [29], [31]–[33], [37] implement fundamentally different encoding schemes. Due to the simplicity of encoders, these designs are faster and more energy-efficient. However, they suffer significantly in accuracy. For HDC inference, the accelerators [31], [37] are on average 57× and 24× faster and consume 3× and 731× less energy than Tri-HD respectively. However, they achieve 12.7-18.2% and 4.9-9.1% less accuracy respectively as compared to Tri-HD. The remaining work [29], [32], [33] do not support the classification tasks discussed in this paper. Hence, a comparison with these work is beyond the scope of the paper.

We also compare Tri-HD with DNN running on FloatPIM [42] for ISOLET dataset. We observe that Tri-HD is 221× (2132×) faster and 1.9× (3.6×) more energy-efficient than FloatPIM with 32-bit fixed point (16-bit bfloat) data representation, while providing 1.7% inference accuracy loss. The speedup is the result of simpler operations in Tri-HD as compared to vector-matrix multiplications in FloatPIM.

E. Tri-HD with Approximate Similarity

Fig. 11 shows the impact of quantization in Tri-HD on the classification accuracy. We show three different policies: the full-precision baseline, round before (pre-round), and round after (post-round), as explained in Section V-A.

1) Accuracy

Fig. 11 shows that the prediction accuracy increases at a diminishing rate as the number of dimension increases. This is true for all five datasets, and for all three rounding policies we used. It is noticeable that both the post-round policy and pre-round policy perform no worse than the reference policy. In fact, on average, Tri-HD using the pre (post) policy, is able to achieve 0.52% (0.36%) better accuracy when using $D = 10,000$.

Fig. 12 shows the impact of the number of retrain iterations and learning rate on the classification accuracy. The plot on the left shows the change in accuracy during retraining on the UCIHAR dataset. This demonstrates that even in the presence of inaccurate computations from quantizing the accumulation

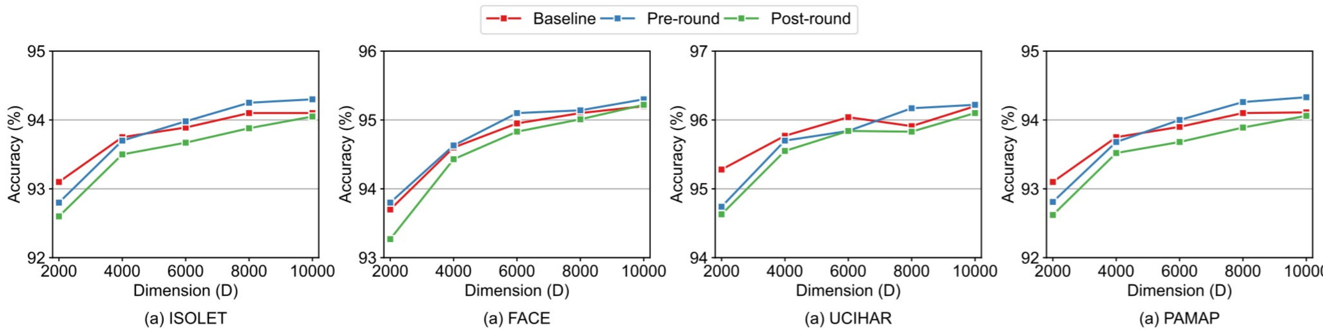


Fig. 11: Impact of HD dimension reduction and rounding approaches on the classification accuracy for different datasets.

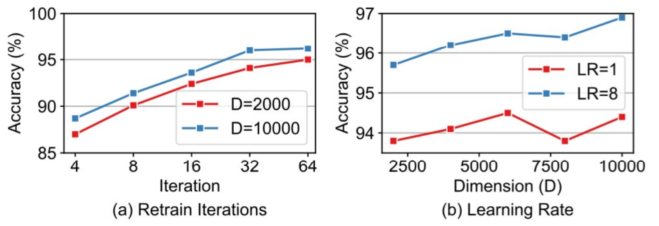


Fig. 12: Impact of number of (a) retraining iterations and (b) learning rate on the application's classification accuracy on data set UCIHAR without rounding.

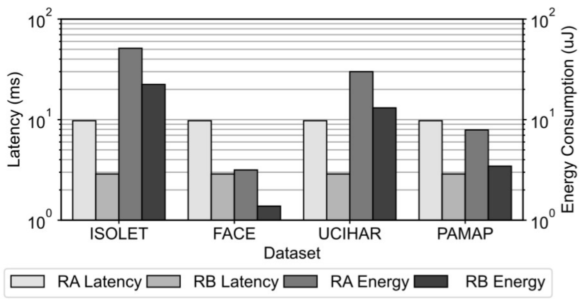


Fig. 13: Latency and energy consumption per similarity check in Tri-HD with different rounding approaches. RA stands for rounding after multiplication and RB stands for rounding before multiplication.

to powers of 2, HD is still able to improve accuracy by a significant amount during retraining. This shows that HD is robust to noise. This property can further be exploited to increase the efficiency of the design as discussed in Section VI-F. The plot on the right demonstrates the difference in accuracy after retraining utilizing different learning rates with varying dimensionality. The optimal learning rate may be different for each dataset. Here we experimentally observed that, as shown by this graph, a learning rate of 8.0 gave close to the best results on average. The learning rate determines how strongly each misprediction during retraining effects the model being trained.

2) Performance and Energy

Fig. 13 shows the latency and energy consumption of a similarity check (dot product) for the two rounding approaches. As expected, similarity metric with rounding before multiplication is on average 3.4× faster and 2.3× more energy-efficient than similarity with rounding after multiplication because it avoids the comparatively slow and high energy consuming in-memory multiplication. For rounding before multiplication, in-memory search happens twice, once for the input and the other for the shifted output. While in the case of rounding after, in-memory search is performed only for the output. We also observe that the number of classes has negligible effect on the latency of similarity check. Whereas, the energy consumption increases almost linearly with the number of classes because Tri-HD needs to check the similarity of an input vector with all the class hypervectors.

F. Tri-HD Energy-Accuracy Trade-off

HD computing works based on the pattern of neural activity which are in high-dimensional space. In theory, the dimensional of the hypervector should be large enough (e.g., $D = 10,000$) to ensure the randomly generated base hypervectors are nearly orthogonal. However, HD computing shows robustness to scaling the hypervector dimensions. Fig. 11 shows the HD accuracy when the hypervector dimension scales from 2000 to 10,000. The result shows that for all applications the HD can provide the similar accuracy as 10,000 when the hypervector dimension scales to 8000. In addition, in reducing the hypervector dimensions from 10000 to 2000, HD loses on average only 1.6% in accuracy.

Tri-HD can exploit the robustness of HD to dimensionality in order to reduce the computation cost. Fig. 14 shows the latency and energy consumption of running HD classification in memory. Our evaluation shows that reducing the hypervector dimension reduces Tri-HD energy consumption. This efficiency comes from the less number of class elements and operations that HD needs to store and process in lower dimension. Our result shows that Tri-HD memory requirement decreases linearly with the hypervector dimensions. For example, HD with D=2000 dimensions consumes 78% lower energy. Note that the latency of Tri-HD does not change with the hypervector dimensions. In fact, Tri-HD is designed to perform bit parallel operations where all computations can be parallelized across different dimensions.

There is a trade-off between the accuracy and efficiency when the hypervector dimension reduces. The results are relative to Tri-HD architecture running the baseline HD with $D = 10,000$ dimensions. Let the quality loss, ΔE, be defined as the difference between the HD classification accuracy in low dimension and $D = 10,000$. When our design ensures 0.5% quality loss ($\Delta E = 0.5\%$), the Tri-HD can provide 25% energy efficiency compared to the baseline model. Similarly, ensuring quality loss of less than 1% (2%), Tri-HD energy and memory

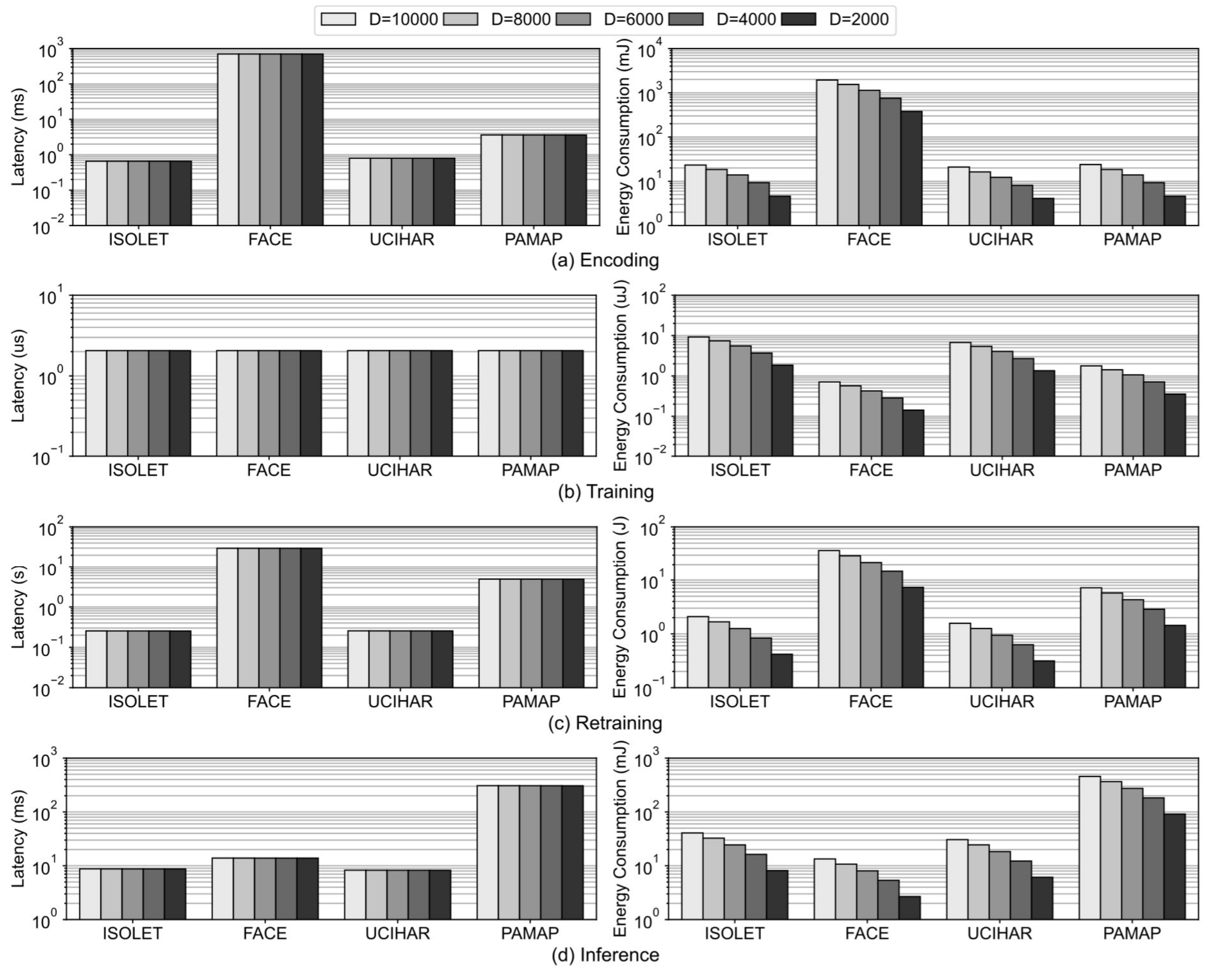


Fig. 14: Latency and energy consumption of Tri-HD for different dimensions.

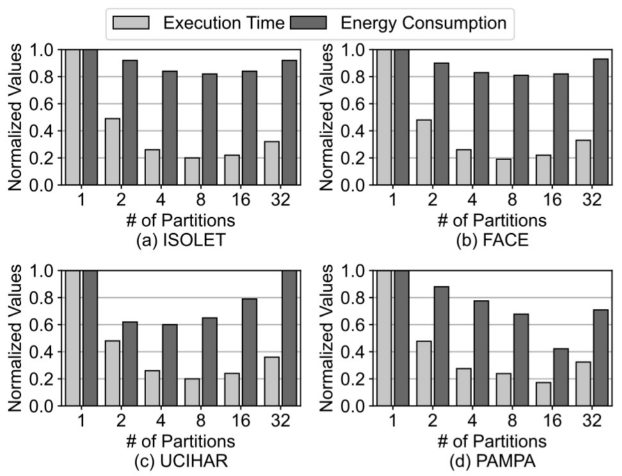


Fig. 15: Impact of number of partitions on the energy consumption and execution time of Tri-HD encoding running different HD applications.

efficiency further improve by 65% and 78% receptively.

G. Tri-HD Parallelism

The HD efficiency depends on the amount of parallelism which we can apply to different modules. The most area efficient method for HD encoding is to store all ID and level hypervectors in a single memory partition and perform XOR operation between each ID and corresponding level in series. This method serially processes the features and its performance is directly related to the number of features. Our design parallelizes the encoding module by partitioning the memory block as discussed in Section IV. For instance, HD using two memory partitions can process two features at the same time. In the best case, the number of memory partitions can be equal to the number of features that exist in application. For example, for ISOLET with 617 features, the encoding operation can be fully parallelized by dividing the memory block into 617 partitions. Each memory partition computes the XOR operation of ID and one of the level hypervectors, which is selected depending on the feature value. There are two overheads of using multiple memory partitions (i) The effective memory requirement increases, since each partition needs to replicate the level hypervectors and assign rows to perform computation and store the XOR result. (ii) All XOR results need to be written in a single memory in order to add together and generate an encoded data. This write operation needs to perform sequentially and degrades the efficiency of using multiple memory block.

Fig. 15 compares the impact of number of memory partitions on the energy efficiency, execution time and memory requirement of HD computing. It shows that increasing the number of partitions at first improves the performance and energy efficiency of the computation. However, it results in less efficiency when the number of partitions surpasses 8. For example, encoding ISOLET with 16 partitions is 10% slower than encoding with 8 partitions. For more than 8 partitions, the cost of combining the results from different partitions exceeds the benefits provided by parallelism due to partitions.

VII. Conclusion

In this paper, we proposed Tri-HD, the first ReRAM PIM architecture to implement the complete HD computing-based classification pipeline for non-binary data. Our design utilizes a novel distance metric that is PIM-friendly and provides similar application accuracy as the more complex baseline metric. Our proposed architecture is enabled in PIM by fast and energy-efficient in-memory logic operations. We further increase the amount of in-memory parallelism by using in-block switches, which segment the bitlines to make parallel operations independent of each other. Our evaluation shows that for all applications tested using HD, Tri-HD provides on average 434× (2170×) speedup and consumes 4114× (26019×) less energy as compared to the CPU while running end-to-end HD training (inference). Tri-HD also achieves at least 2.2% higher classification accuracy than all existing PIM-based HD designs.

Acknowledgments

This work was supported in part by , the Center for Processing with Intelligent Storage and Memory (PRISM) SRC grant number 2023-JU-3135, CoCoSys, centers in JUMP 2.0, an SRC program sponsored by DARPA, TILOS AI Research Institute (NSF CCF-2112665), and also NSF grants # 1527034, #1730158, #1826967, #1911095, and #2003279.

References

[1] C. Perera *et al.*, "Context aware computing for the internet of things: A survey," *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 414–454, 2013.

[2] J. Gubbi *et al.*, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] A. Rahimi *et al.*, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *International Symposium on Low Power Electronics and Design*, pp. 64–69, 2016.

[4] Y. Kim *et al.*, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, pp. 1–6, 2018.

[5] S. Gupta *et al.*, "Thrifty: training with hyperdimensional computing across flash hierarchy," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2020.

[6] W. Xu *et al.*, "Hypermetric: Robust hyperdimensional computing on error-prone memories using metric learning," in *IEEE 41st International Conference on Computer Design (ICCD)*, pp. 243–246, IEEE, 2023.

[7] W. Xu *et al.*, "Hyperspec: Ultrafast mass spectra clustering in hyperdimensional space," *Journal of Proteome Research*, 2023.

[8] Z. Zou *et al.*, "Biohd: an efficient genome sequence search platform using hyperdimensional memorization," in *Annual International Symposium on Computer Architecture*, pp. 656–669, 2022.

[9] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[10] G. H. Loh *et al.*, "A processing in memory taxonomy and a case for studying fixed-function pim," in *Workshop on Near-Data Processing (WoNDP)*, pp. 1–4, 2013.

[11] Q. Guo *et al.*, "Ac-dimm: associative computing with stt-mram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 189–200, 2013.

[12] Q. Guo *et al.*, "A resistive tcam accelerator for data-intensive computing," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 339–350, 2011.

[13] L. Yavits *et al.*, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2014.

[14] S. Gupta *et al.*, "Rapid: A reram processing in-memory architecture for dna sequence alignment," in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, 2019.

[15] A. Siemon *et al.*, "A complementary resistive switch-based crossbar array adder," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 5, no. 1, pp. 64–74, 2015.

[16] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6, 2016.

[17] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[18] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, 2017.

[19] Nejatollahi *et al.*, "Cryptopim: in-memory acceleration for lattice-based cryptographic hardware," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

[20] S. Gupta *et al.*, "Scrimp: A general stochastic computing architecture using reram in-memory processing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1598–1601, 2020.

[21] S. Kvatinsky *et al.*, "MAGIC – memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[22] S. Kvatinsky *et al.*, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2013.

[23] A. Haj-Ali *et al.*, "Efficient algorithms for in-memory fixed point multiplication using magic," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2018.

[24] J. Borghetti *et al.*, "'memristive'switches enable 'stateful'logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[25] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *Proceedings of the International Conference on Computer-Aided Design*, p. 55, 2018.

[26] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.

[27] M. Imani *et al.*, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, 2017.

[28] M. Imani *et al.*, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.

[29] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 445–456, 2017.

[30] M. Zhou *et al.*, "Thermal-aware design and management for search-based in-memory acceleration," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[31] S. Datta *et al.*, "A programmable hyper-dimensional processor architecture for human-centric iot," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 439–452, 2019.

[32] T. F. Wu *et al.*, "Hyperdimensional computing exploiting carbon nanotube fets, resistive ram, and their monolithic 3d integration," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 11, pp. 3183–3196, 2018.

[33] G. Karunaratne *et al.*, "In-memory hyperdimensional computing," *arXiv preprint arXiv:1906.01548*, 2019.

[34] H. Li *et al.*, "Hyperdimensional computing with 3d vrram in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *IEEE International Electron Devices Meeting (IEDM)*, pp. 16–1, 2016.

[35] J. Liu *et al.*, "Hdc-im: Hyperdimensional computing in-memory architecture based on rram," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 450–453, 2019.

[36] S. Hamdioui *et al.*, "Applications of computation-in-memory architectures based on memristive devices," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 486–491, 2019.

[37] M. Imani *et al.*, "Searchd: A memory-centric hyperdimensional computing with stochastic training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[38] "Uci machine learning repository: Isolet dataset." http://archive.ics.uci.edu/ml/datasets/ISOLET, 1994.

[39] G. Griffin *et al.*, "Caltech-256 object category dataset," 2007.

[40] "Uci machine learning repository: Har dataset." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities, 2012.

[41] A. Reiss and D. Stricker, "Creating and benchmarking a new dataset for physical activity monitoring," in *The 5th International Conference on PErvasive Technologies Related to Assistive Environments*, p. 40, 2012.

[42] M. Imani *et al.*, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pp. 802–815, 2019.

[43] N. Hajinazar *et al.*, "Simdram: a framework for bit-serial simd processing using dram," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 329–345, 2021.

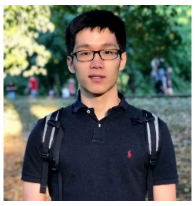
[44] J. Morris *et al.*, "Hydrea: Utilizing hyperdimensional computing for a more robust and efficient machine learning system," *ACM Transactions on Embedded Computing Systems*, vol. 21, no. 6, pp. 1–25, 2022.

[45] M. Imani *et al.*, "Nvquery: Efficient query processing in nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 628–639, 2018.

[46] K. Prabhu *et al.*, "Chimera: A 0.92-tops, 2.2-tops/w edge ai accelerator with 2-mbyte on-chip foundry resistive ram for efficient training and inference," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 4, pp. 1013–1026, 2022.

[47] Q. Luo *et al.*, "Nb 1-x o2 based universal selector with ultra-high endurance (¿ 10 12), high speed (10ns) and excellent v th stability," in *Symposium on VLSI Technology*, pp. T236–T237, IEEE, 2019.

[48] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.



Weihong Xu received the B.E. degree in information engineering and M.E. in information and communication engineering from Southeast University, Nanjing, China, in 2017 and 2020. He is currently a fourth-year Ph.D. student in Computer and Computer Engineering at the University of California San Diego, La Jolla, CA, USA. His research interests include in-memory and in-storage architecture for deep learning, bioinformatics, and hyperdimensional computing.

Saransh Gupta received his Ph.D. degree from the University of California, San Diego, La Jolla, CA, USA in 2021. He received his B.E. (Hons) in Electrical and Electronics Engineering from Birla Institute of Technology & Science, Pilani - K.K. Birla Goa Campus in 2016 and M.S. in Electrical and Computer Engineering from University of California San Diego in 2018. His research interests include circuit, architecture, and system level aspects of emerging computing paradigms.

Justin Morris received the B.S. degree, in 2018, and the joint Ph.D. degree from the University of California at San Diego and San Diego State University, in 2022. He had the pleasure of being advised by both Dr. Tajana Rosing and Dr. Baris Aksanli. Before starting the Ph.D. degree, he was an Undergraduate Student Researcher with the University of California at San Diego. He took a gap year focusing on research, while he applied for the Ph.D. degree to stay in Dr. Tajana Rosing's research group with the System Energy Efficiency Laboratory. During both the undergraduate and graduate degrees, he was actively engaged in teaching. He was a Tutor, TA, and also an Instructor with the University of California at San Diego. Recently, he joined California State University San Marcos as an Assistant Professor in computer engineering, in Fall 2022.

Xincheng Shen received the Bachelor degree in 2021 and M.S. degree in 2023 from the Department of Computer Science, UC San Diego. He is currently a software engineer for Vista in San Mateo, California, United States.

Mohsen Imani received the Ph.D. degree from the Department of Computer Science, UC San Diego. He is currently an Assistant Professor with the Department of Computer Science, UC Irvine. He is also the Director of the Bio-Inspired Architecture and Systems (BIASLab). His contribution has led to a new direction on brain-inspired hyperdimensional computing that enables ultra-efficient and real-time learning and cognitive support. His research was also the main initiative in opening up multiple industrial and governmental research programs. His research has been recognized with several awards, including the Bernard and Sophia Gordon Engineering Leadership Award, the Outstanding Researcher Award, and the Powell Fellowship Award. He also received the Best Doctorate Research from UCSD and several best paper nomination awards at multiple top conferences, including the Design Automation Conference (DAC) in 2019 and 2020, the Design Automation and Test in Europe (DATE) in 2020, and the International Conference on Computer-Aided Design (ICCAD) in 2020. Furthermore, he received the Best Paper Award in DATE 2022 Conference.

Baris Aksanli is currently an Assistant Professor with the Electrical and Computer Engineering Department, San Diego State University, San Diego, CA, USA. Previously, he was a Postdoctoral Researcher at the Computer Science and Engineering Department, University of California San Diego. As a Researcher, his affiliations include the Multi Scale Systems Center (MuSyC), the TerraSwarm Research Center, and the Center for Networked Systems (CNS); and the collaborators of his projects include Google, Microsoft, Panasonic, Intel, and IBM. His research interests include energy efficiency and peak power management of large-scale systems, such as data centers and smart grids, efficient battery usage in data centers and residential houses, battery lifetime modeling, cost and energy aware automation of residential houses, learning techniques to enhance user behavior modeling and context extraction, house/building/data center, and grid interaction.

Tajana Rosing received her Ph.D. degree from Stanford University, Stanford, CA, USA, in 2001. She is a Professor, a Holder of the Fratamico Endowed Chair, and the Director of System Energy Efficiency Laboratory, University of California at San Diego, La Jolla, CA, USA. From 1998 to 2005, she was a full-time Research Scientist with HP Labs, Palo Alto, CA, USA, while also leading research efforts with Stanford University, Stanford, CA, USA. She was a Senior Design Engineer with Altera Corporation, San Jose, CA, USA. She is leading a number of projects, including efforts funded by DARPA/SRC JUMP 2.0 PRISM program with focus on design of accelerators for analysis of big data, DARPA and NSF funded projects on hyperdimensional computing, and SRC funded project on IoT system reliability and maintainability. Her current research interests include energy-efficient computing, cyber–physical, and distributed systems.