# A Computing-in-Memory-Based One-Class Hyperdimensional Computing Model for Outlier Detection

Ruixuan Wang, *Graduate Student Member, IEEE*, Sabrina Hassan Moon, *Student Member, IEEE*, Xiaobo Sharon Hu, *Fellow, IEEE*, Xun Jiao, *Member, IEEE*, and Dayane Reis, *Senior Member, IEEE*

***Abstract*—In this work, we present ODHD, an algorithm for outlier detection based on hyperdimensional computing (HDC), a non-classical learning paradigm. Along with the HDC-based algorithm, we propose IM-ODHD, a computing-in-memory (CiM) implementation based on hardware/software (HW/SW) codesign for improved latency and energy efficiency. The training and testing phases of ODHD may be performed with conventional CPU/GPU hardware or our IM-ODHD, SRAM-based CiM architecture using the proposed HW/SW codesign techniques. We evaluate the performance of ODHD on six datasets from different application domains using three metrics, namely accuracy, F1 score, and ROC-AUC, and compare it with multiple baseline methods such as OCSVM, isolation forest, and autoencoder. The experimental results indicate that ODHD outperforms all the baseline methods in terms of these three metrics on every dataset for both CPU/GPU and CiM implementations. Furthermore, we perform an extensive design space exploration to demonstrate the tradeoff between delay, energy efficiency, and performance of ODHD. We demonstrate that the HW/SW codesign implementation of the outlier detection on IM-ODHD is able to outperform the GPU-based implementation of ODHD by at least 331.5×/889× in terms of training/testing latency (and on average 14.0×/36.9× in terms of training/testing energy consumption).**

***Index Terms*—Hyperdimensional computing, outlier detection, computing-in-memory, hardware/software codesign.**

## I. Introduction

OUTLIER detection, also referred to as anomaly detection, is a crucial technique utilized in various application domains like medical diagnosis, Internet-of-Things (IoT), and financial fraud detection. Outliers are generally extreme or out-of-distribution values in a dataset that deviate from other samples or an observation that does not fit the overall pattern. These outliers typically suggest measurement variability, experimental errors, or novelty. In machine learning, outliers in the training or testing set may cause failure in the detection or classification tasks. Additionally, in recent times, cyber attackers deliberately fabricate outliers, posing a threat to the security of cyber-physical systems.

Over the years, researchers continue to design robust solutions to detect outliers efficiently and effectively, where statistical methods and machine learning methods are the two most popular types of solutions. Statistical methods include parametric methods such as Gaussian mixture model (GMM) methods [1] and non-parametric methods such as kernel density estimation methods [2]. While statistical methods are mathematically well explainable, fast to evaluate, and easy to implement, their results could be unreliable for practical applications due to their dependency on assumptions of a specific distribution model.

Recently, using machine learning techniques for outlier detection has witnessed a significant surge. Among the most effective methodologies, one-class support vector machine (OCSVM), isolation forest, and autoencoder-based neural network approaches are the most notable. OCSVM, which is a variant of the conventional SVM, distinguishes outliers from inliers by maximizing the margin [3]. The isolation forest method, on the other hand, utilizes an ensemble model consisting of isolation trees, with outliers being more vulnerable to isolation and having shorter traversal path lengths [4]. The autoencoder-based method is a novel unsupervised learning approach that uses neural networks to reconstruct data samples, identifying outliers based on the reconstruction errors [5]. These traditional machine learning-based methods can achieve accurate outlier detection but may lack consideration for computation and energy efficiency.

In this paper, we present a novel approach for outlier detection based on hyperdimensional computing (HDC). HDC is an emerging computing paradigm inspired by the human brain circuitry that exhibits high-dimensionality and fully distributed holographic representation [6], [7]. HDC represents data samples using high-dimensional hypervectors, typically dimension $D = 10,000$, which can be generated, manipulated, and compared to perform learning tasks. Compared to deep neural networks (DNNs), HDC offers several advantages, including smaller model size, lower computational cost, and one/few-shot learning, making it an attractive alternative, particularly

Manuscript received 9 May 2023; revised 12 February 2024; accepted 21 February 2024. Date of publication 1 March 2024; date of current version 10 May 2024. This work was supported in part by the College of Engineering at USF, the U.S. NSF under Grant 2202310, and in part by ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR. Recommended for acceptance by T. Adegbija. *(Ruixuan Wang and Sabrina Hassan Moon contributed equally to this work.) (Corresponding authors: Xun Jiao; Dayane Reis.)*

Ruixuan Wang and Xun Jiao are with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA 19085 USA (e-mail: xun.jiao@villanova.edu).

Sabrina Hassan Moon and Dayane Reis are with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620 USA (e-mail: dayane3@usf.edu).

Xiaobo Sharon Hu is with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556 USA.

Digital Object Identifier 10.1109/TC.2024.3371782

for low-cost computing platforms [7]. HDC has demonstrated promising results in diverse applications such as computer vision [8].

Specifically, we propose **ODHD**, which is a novel one-class HDC-based outlier detection method using a positive-unlabeled (P-U) learning structure [9]. Our approach is based on the simple yet reasonable assumption that a single hypervector (HV) can represent the abstract information of all inlier samples, which can be distinguished from outlier samples represented in HVs. Although HDC has been extensively studied for supervised learning tasks such as classification in various domains [7], there is limited research on using HDC for other tasks. Furthermore, recognizing the memory-centric computing properties of **ODHD**, we propose a hardware/software codesign implementation of **ODHD**'s both training and inference phases on top of a computing-in-memory (CiM) architecture (**IM-ODHD**). CiM can curtail the memory access bottleneck by leveraging parallelism inside the memory array structure, which enables computation at the bitline level along several current paths simultaneously. CiM has emerged as one of the most promising approaches for signal processing, optimization, deep learning and stochastic computing [10]. Our experimental results indicate that CiM can significantly accelerate the **ODHD** algorithm and deliver superior energy efficiency.

Built on top of our previous study in [11], this paper makes the following contributions:

1) We introduce **ODHD**, a novel one-class outlier detection method based on HDC and P-U learning. Our approach forms a high-dimensional representation of inlier samples and is a viable alternative to existing outlier detection approaches.
2) We develop a comprehensive pipeline for **ODHD** algorithm. First, we map all inliers samples to a high-dimensional space and create a one-class HV to represent the abstract information of inliers. Next, we propose a confidence-based method to automatically compute a threshold that is used for outlier detection. During testing, we compute the similarity between the unseen testing sample and the one-class HV and compare it to the pre-computed threshold to detect outliers.
3) We propose a static random-access memory (SRAM)-based CiM architecture, **IM-ODHD**, to implement **ODHD**. Our CiM architecture leverages customized elements (sense amplifiers), mat-level row/column decoders, logarithmic bit shifters, etc., to attain reduced latency and increased parallelism supporting different parameters of HDC, such as a number of dimensions and hypervector seeds.
4) We apply a hardware/software codesign approach to further improve the functionality of **IM-ODHD** on CiM architecture. Specifically, we adjust the algorithm-level design of **ODHD** and show that the proposed changes speed up the runtime of both training and inference with insignificant accuracy loss.

To evaluate **ODHD**, we use six datasets from the Outlier Detection Datasets (ODDS) Library [12] and compare our approach with baseline methods such as OCSVM, isolation forest, autoencoder, and HDAD. The comprehensive evaluation results show that **ODHD** outperforms all the baseline methods on all six datasets in all metrics, including accuracy, F1 score, and ROC-AUC with both CPU/GPU and CiM implementations. Furthermore, after the hardware/software codesign adjustment, we demonstrate that our hardware/software codesign implementation of **IM-ODHD** is able to outperform the GPU-based implementation of the same algorithm by at least 293×/ 419× in terms of training/testing latency (and on average 16.0×/15.9× in terms of training/testing energy consumption). Our study demonstrates the effectiveness of **ODHD** in the realm of both software and hardware and highlights its potential for research in outlier detection.

The rest of the paper is structured as follows. In Section II, we discuss the fundamentals of HDC and CiM. Section III introduces the **ODHD** algorithm. An SRAM-based CiM architecture for **ODHD** (**IM-ODHD**) is proposed in Section IV. We evaluate the performance, energy, and latency of **ODHD** and **IM-ODHD** in Section V. Related works are presented and discussed in Section VI. Finally, Section VII concludes the paper.

## II. Background

Here, we discuss the mathematical foundations and operations of HDC. Furthermore, we discuss the basics of CiM.

### A. Hyperdimensional Computing

***Basic HDC Component:*** Hypervectors (HVs) are the fundamental components of HDC. An HV is a holographic and high-dimensional vector with independent and identically distributed (i.i.d.) elements. The HV with $D = d$ dimensions is denoted as $\overrightarrow{H} = \langle h_1, h_2, \ldots, h_d \rangle$. In this paper, we employ bipolar HVs, which means each element in an HV is either $-1$ or 1 [7].

In HDC, HVs are used as the information representation in different scales and levels, such as embedding new information or aggregating existing information. To measure the correlation between information representation, we use cosine distance to measure the similarity of information between two HVs, as shown in Eq. 1. Moreover, one property of HVs is, when the dimensionality is sufficiently high (e.g., $D = 10,000$), HVs are quasi-orthogonal whereas any two random bipolar HVs are nearly orthogonal [6].

$$\delta(\overrightarrow{H_x}, \overrightarrow{H_y}) = \frac{\overrightarrow{H_x} \cdot \overrightarrow{H_y}}{||\overrightarrow{H_x}|| \times ||\overrightarrow{H_y}||} = \frac{\sum_{i=1}^{d} h_{xi} \cdot h_{yi}}{\sqrt{\sum_{i=1}^{d} {h_{xi}}^2} \cdot \sqrt{\sum_{i=1}^{d} {h_{yi}}^2}} \tag{1}$$

***Basic HDC Operations:*** HDC supports three basic arithmetic operations including bundling, binding and permutation, as illustrated in Eq. 2. Additions and multiplications both take two input HVs as operands and perform **element-wise** add or multiply operations. Permutation takes one HV as the input operand and performs **cyclic rotation**.

$$\begin{aligned} bundling(\overrightarrow{H_x} + \overrightarrow{H_y}) &= \langle h_{x1} + h_{y1}, h_{x2} + h_{y2}, \ldots, h_{xd} + h_{yd} \rangle \\ binding(\overrightarrow{H_x} * \overrightarrow{H_y}) &= \langle h_{x1} * h_{y1}, h_{x2} * h_{y2}, \ldots, h_{xd} * h_{yd} \rangle \\ permutation^1(\overrightarrow{H}) &= \langle h_d, h_1, h_2, \ldots, h_{d-1} \rangle \end{aligned} \tag{2}$$

All three operations preserve the dimensionality of the input HVs, i.e., the input HVs and the output HVs have the same dimension. Considering the three main operations, bundling adds the same type of information, binding aggregates various types of information together to generate new information, and permutation reflects the spatial or temporal changes, such as time series or spatial coordinates [6].

### B. Computing-in-Memory

The limited processor-memory bandwidth significantly impacts a system's performance. Computing-in-memory (CiM) performs the logic and memory operations associated with a given task within the memory boundaries. CiM exploits the large, internal bandwidth of memory to achieve parallelism, which reduces latency and saves energy due to fewer external memory references. CiM architectures may target either general-purpose or application-specific designs, as described below.

*1) Application-Specific CiM Designs:* Examples of application-specific CiM designs include the in-memory computation of dot-products with crossbars [13] and search with non-volatile ternary content addressable memories (TCAMs) [14], which are suitable for performing nearest neighbor operations. The majority of CiM implementations for HDC rely on application-specific designs based on crossbars and TCAMs. These CiM architectures typically employ emerging memory technologies (EMTs) such as Ferroelectric Field-Effect Transistors (FeFETs) (e.g., [15]) and Resistive Random-Access Memories (ReRAMs) (e.g., [16]). EMTs have great potential for high-density and low-power implementations of CiM-based HDC. For instance, CiM improves energy consumption by 826× and latency by 30× for a classification task with HDC when compared to a GPU baseline [15]. However, as development on EMTs is still in its early phases, there is a lack of large-scale solutions for CiM-based HDC that can be promptly integrated into real systems. Furthermore, much of the computation with EMTs in application-specific CiM designs occurs in the analog domain, which limits the bit precision due to the physical limits of the EMTs, as well as the errors induced by circuit components such as the analog-to-digital converters (ADCs). The limited precision makes it challenging to match software accuracies.

*2) General-Purpose CiM Designs:* General purpose CiM (GPCiM) designs support logic and arithmetic operations that can benefit different applications as they can be used to implement different algorithms [17]. In this work, we propose a GPCiM architecture that is capable of performing all the operations needed by HDC-based outlier detection. The algorithmic flow for outlier detection with HDC running on our CiM architecture is presented in Section III. The CiM architecture of **ODHD**, which we name as **IM-ODHD**, is described in Section IV, along with a hardware/software codesign approach for **ODHD** that allows for mapping of the outlier detection algorithm onto **IM-ODHD**. **IM-ODHD** operates in the digital domain, with customized sense amplifiers, local copy drivers, and bit shifters, achieving high parallelism with multiple subarrays operating simultaneously to perform in-memory operations. The circuits employed in **IM-ODHD** are illustrated in part *b* of Fig. 2 and described below.

***Word Line Decoders [18]:*** The simultaneous sensing of multiple rows in an SRAM subarray is possible by lowering the word line voltage to bias against the write of the SRAM. As shown in Fig. 2, to leverage double sensing, our design implements two-word line decoders in the same PE to simultaneously activate two rows for performing computation between them.

***Customized Sense Amplifier (CSA) [19]:*** Once the subarray rows are activated, voltage drops on the memory bitlines (and negated bitlines), while the actual values depend on the operands stored in the SRAM. The voltage drop can be sensed with CSA, which will generate the results for different bitwise logic operations (e.g., AND, OR, XOR) and arithmetic between the two rows of data. The output of the CSA depends on the desired operation, which is selected with an internal multiplexer circuit.

***Logarithmic Bit Shifters [18]:*** The output of the CSA is passed as input to the logarithmic bit shifter, which can shift the output to the left or right. The number of bit positions by which a binary number is shifted left or right is determined by the logarithm of a shift amount (i.e., the shift mask in our circuit). The main advantage of this circuit over a traditional linear bit shifter is that it can perform larger shifts in a single clock cycle, rather than shifting one-bit position at a time. The logarithmic bit shifter in **IM-ODHD** can accelerate permutations and divisions by powers-of-two in **ODHD**. In the logarithmic bit shifter used in **IM-ODHD** (shown in Fig. 2), a 3-bit shift mask goes into each PE to configure 0-3 bit shifts to the left or right (1 bit of the mask determines the shift direction, while the other two bits are for the shift amount). Shift amounts larger than 3 bits are possible through a multi-step approach.

***Write and Copy Drivers [20]:*** Memory write drivers are circuits that play a crucial role in writing data into the 6T-SRAM memory cells. These drivers work by amplifying the signals from an external memory controller to generate the required voltage levels for writing data into the memory cells through the bit lines. In the context of the **ODHD**, write drivers are utilized to write both the initial HVs and intermediated HVs from the mat-level registers into the PEs. Copy drivers are another type of circuitry used in **ODHD** specifically for copying the results of CiM operations, such as bitwise logic, addition, or right/left bit shift, to a designated address within the same PE. To ensure efficient and effective copying, copy drivers are placed in alignment with the CSA columns in the subarray.

## III. ODHD: Algorithm

We leverage the mathematical properties of HDC to develop a novel one-class HDC-based outlier detection algorithm, which essentially learns an abstract representation of inlier samples and then performs one-class classification-based outlier detection. In **ODHD**, the outlier detection process is based on a P-U learning structure [9], which means we use only inlier samples for training and test on a testing set (may contain

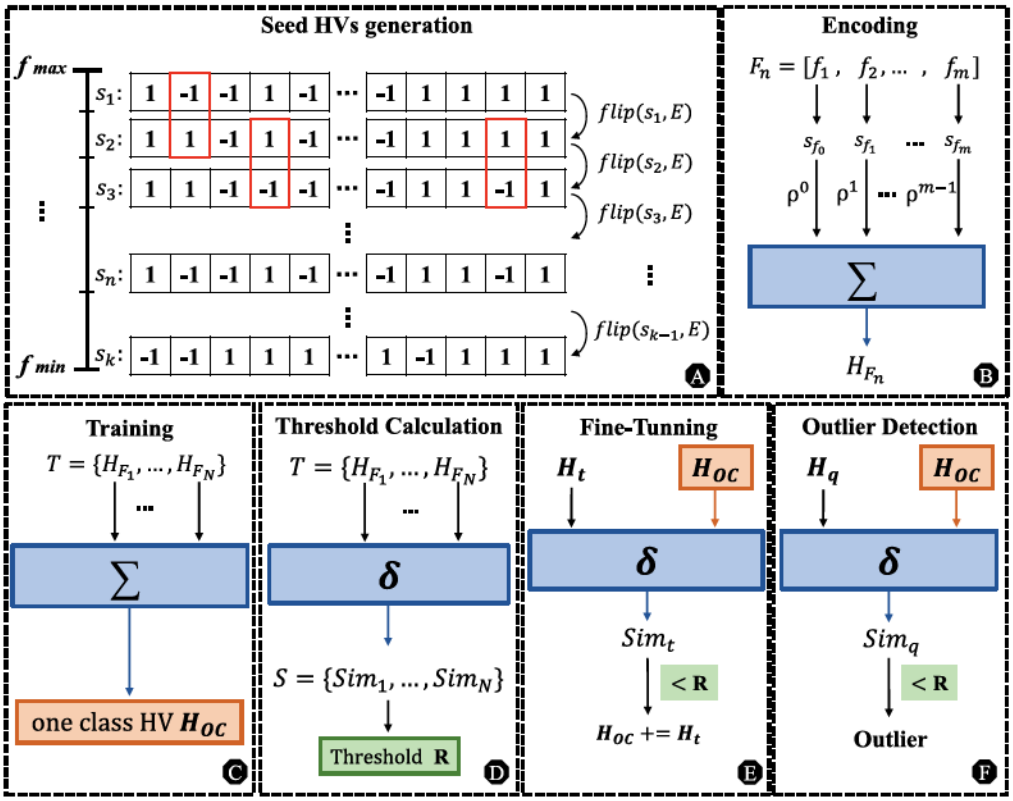



Fig. 1. The algorithmic flow of **ODHD** with six key phases.

both inliers and outliers) without the information of labels. The one-class HV we trained contains the information from all the patterns of inlier (training) samples. For inference, we detect whether a query HV conforms to the one-class HV according to cosine similarity. In **ODHD**, we utilize a confidence-based procedure to calculate a threshold based on training HVs. If the cosine similarity between a query HV and one-class HV is lower than the threshold, the query HV will be detected as an outlier. Fig. 1 illustrates the whole algorithmic flow of **ODHD**, which is divided into six key phases: Seed HV Generation, Encoding, Training, Threshold Calculation, Fine-Tuning, and Outlier Detection. We describe each phase of the algorithm in detail in the following sections.

### A. Seed HVs Generation

As the first step, we need to generate seed HVs so that we can encode the raw sample features into HVs. As noted previously, each HV is a high-dimensional vector with i.i.d elements [6]. We employ an HV-generating method consistent with the one in [21] to create $k$ seed HVs that can support later encoding, which is more computationally efficient compared to randomly generating $k$ random HVs straightforwardly while preserving the orthogonality of HVs. As part A of Fig. 1 illustrates, we initiate a random bipolar $D$-dimension HV, $\overrightarrow{s_1}$, and then generate all the seed HVs by randomly flipping $E = D/2k$ elements. Specifically, $k$ is a configurable parameter depending on how we discretize the input data. Consequently, a set of seed HVs $\{\overrightarrow{s_1}, \overrightarrow{s_2}, \ldots, \overrightarrow{s_k}\}$ is generated. For the following encoding procedure, assume for a specific dataset, we have each feature vector with $m$ feature elements $\overrightarrow{F_n} = \langle f_1, f_2, \ldots, f_m \rangle$. According to the training set, we can capture the minimum and the maximum values of each feature value $f_{min}$ and $f_{max}$. Then we can discretize the input feature space $(f_{min}, f_{max})$ into $k$ uniform intervals. Thus, each feature value corresponds to a specific interval, and we can map the feature vector into an integer vector for encoding.

### B. Encoding

The encoding step projects the original feature vector into an HV. The encoding process of feature vector $\overrightarrow{F_n} = \langle f_1, f_2, \ldots, f_m \rangle$ is shown in part B of Fig. 1. We first index the seed HV corresponding to each feature value. For example, if the feature element $f_2$ falls into the $5^{th}$ interval among the $k$ intervals, the corresponding seed HV is the $5^{th}$ of the $k$ seed HVs. Then, we employ the permutation operation to embed the information of the feature position into the seed HV. As the permutation operation reflects the spatial change of information, we bundle the information of feature position by deploying a cyclic rotation on each seed HV as shown in Eq. 2. Particularly, we keep the first seed HV un-permuted $(\rho^0(\overrightarrow{s_{f_1}}))$, and for seed HV $\overrightarrow{s_2}$ to $\overrightarrow{s_k}$, we circularly rotate the $i^{th}$ seed HV by $i-1$ elements, i.e., $\rho^{i-1}(\overrightarrow{s_{f_i}})$.

At the end of the encoding process, we aggregate all permuted seed HVs corresponding to all feature values into one HV $\overrightarrow{H_{F_n}}$ representing the entire feature vector $\overrightarrow{F_n}$. Note that if we have 100 inlier samples (i.e., 100 feature vectors) in the training

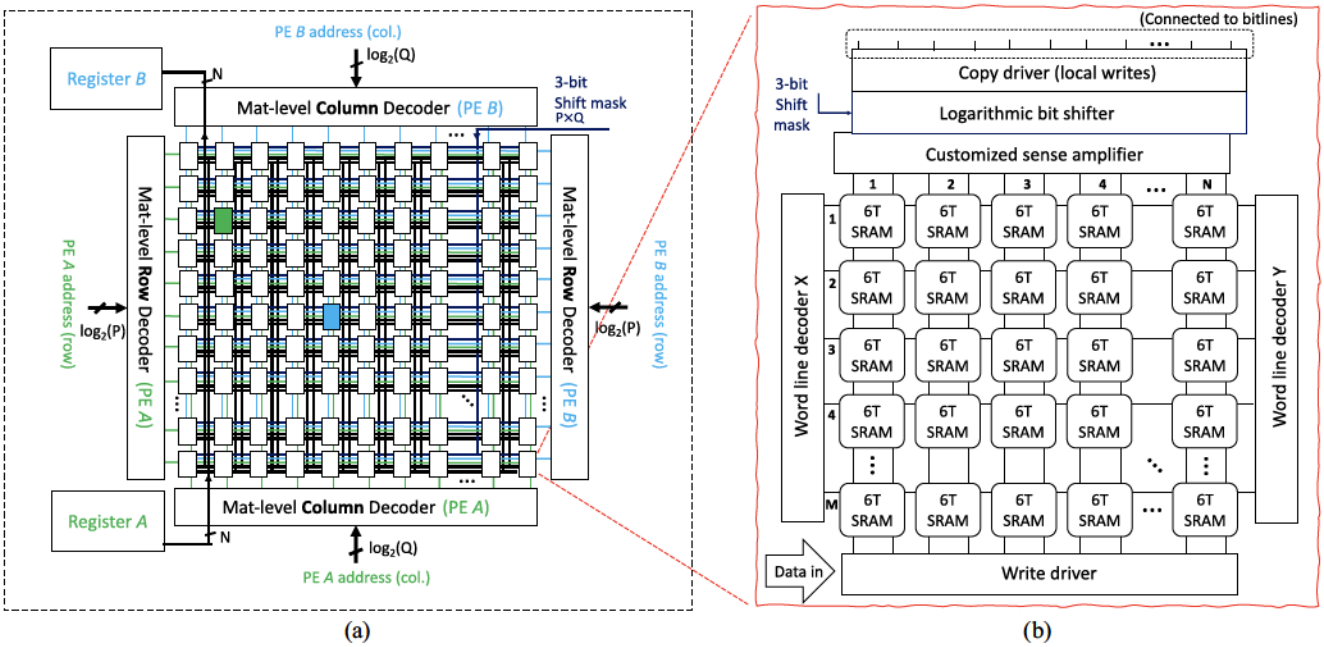



Fig. 2. IMC architecture for outlier detection. (a) $(P \times Q)$ mat-level architecture. The PEs are marked green for source and blue for destination. (b) Detail of one $(M \times N)$ subarray.

dataset, we would have 100 corresponding encoded HVs. The overall encoding process is denoted as Eq. 3.

$$\overrightarrow{H_{F_n}} = \rho^0(\overrightarrow{s_{f_1}}) + \rho^1(\overrightarrow{s_{f_2}}) + \cdots + \rho^{m-1}(\overrightarrow{s_{f_m}}) \tag{3}$$

### C. Training

As part C in Fig. 2 indicates, after encoding all feature vectors in the training set, the training phase generates the one-class HV ($H_{OC}$) of the entire training set, i.e., all inlier samples. Eq. 4 illustrates the process of HDC training, which bundles all the HV representing each inlier feature vector. For example, if there are 100 inlier samples, then the 100 corresponding encoded HVs generated by the encoding process are added together to generate a single one-class HV $H_{OC}$ representing inlier samples or patterns.

$$\overrightarrow{H_{OC}} = \sum_{i=1}^{N} \overrightarrow{H_{F_i}} \tag{4}$$

### D. Threshold Calculation

In ODHD, we propose a confidence-based threshold calculation approach. In order to calculate a threshold to separate inliers and outliers, we measure the cosine similarity between $\overrightarrow{H_{OC}}$ and all training HVs to obtain a similarity array $S$. As part D in Fig. 1 shows, each similarity $Sim_i$ in array $S$ can be considered as the confidence of the training HV to be an inlier sample.

We calculate the mean value $\mu(S)$ and the standard deviation $\sigma(S)$ over all the similarity values in array $S$. We then deploy the threshold estimation strategy shown in Eq. 5, which is established in prior research [5], [22].

$$R = \mu(S) + 2 * \sigma(S) \tag{5}$$

Ultimately, we compute the threshold $R$ based on the confidence of all training HVs. In the outlier detection domain, only the samples with cosine similarity higher than the threshold are determined as an inlier, while all the samples with cosine similarity lower than the threshold are identified as outliers.

### E. Fine-Tuning

After the training phase, we expect that all training HVs should be properly determined as inliers (but this may not be the case). Hence, we perform fine-tuning for ODHD to enhance the performance of outlier detection. The fine-tuning process, shown in part E of Fig. 1, is automatically conducted via predefined rules consisting of two steps: (1) measure the similarity metric between the encoded training HVs and the one-class HV; (2) if this similarity metric falls below the threshold calculated in step (D) of Fig. 1, incorporate the training HV into the one-class HV.

The fine-tuning process acts as an auto-calibration process, and the single parameter that needs to be set by the user is the number of epochs for the fine-tuning process, e.g., in this paper, we executed the fine-tuning process for a total of 10 epochs. Note that we still only use the given training dataset for the fine-tuning process. In each fine-tuning epoch, we feed all the training samples to ODHD. For each training sample, we estimate the cosine similarity $Sim_t$ between the training HV $\overrightarrow{H_t}$ and one-class HV $\overrightarrow{H_{OC}}$. If $Sim_t$ is higher than threshold $R$, which means the estimation is correct, we do not make any changes to $\overrightarrow{H_{OC}}$. However, if $Sim_t$ is lower than $R$, which means ODHD mistakenly considers the inlier sample $t$ as an outlier, we update the one-class HV: we add the misclassified training HV $\overrightarrow{H_t}$ into the one-class HV to update the corresponding information in $\overrightarrow{H_{OC}}$.

### F. Outlier Detection

After we train $\overrightarrow{H_{OC}}$ and obtain threshold $R$ based on the confidences of the training HVs, we deploy the outlier detection on an unseen sample without knowledge of the labels. The outlier detection process is shown as part F in Fig. 1.

During the outlier detection phase, we encode testing sample $q$ into an HV called query HV, $\overrightarrow{H_q}$, following the same encoding process in Eq. 3 based on the same seed HVs. Then we compute the cosine similarity $Sim_q$ between the query HV $\overrightarrow{H_q}$ and the one-class HV $\overrightarrow{H_{OC}}$. In the event that $Sim_q$ is lower than the pre-computed threshold, the sample $q$ will be determined as an outlier.

$$\overrightarrow{H_q} = \begin{cases} Inlier & Sim_q \geq R \\ Outlier & Sim_q < R \end{cases} \tag{6}$$

## IV. IM-ODHD: Hardware

In this section, we first describe **IM-ODHD**, the CiM-based hardware architecture for **ODHD** (Section IV-A). We then discuss our hardware-algorithm codesign effort in adjusting **ODHD** to the **IM-ODHD** hardware (Section IV-B).

### A. GPCiM Architecture

The combination of HDC and CiM can be particularly beneficial since HDC operations involve the manipulation of holographic HVs, which can be performed efficiently in memory. By performing HDC operations in memory using CiM, it is possible to achieve significant improvements in performance and energy efficiency compared to traditional von Neumann architectures (as demonstrated in [15], [16]).

In this work, to implement the HDC-based outlier detection algorithms, we depart from the use of application-specific CiM designs based on NVM. Instead, we design **IM-ODHD** as a general-purpose CiM architecture based on CMOS (i.e., with 6T-SRAMs). The use of an SRAM-based design instead of an NVM-based one leads to several advantages: (1) Our architecture can perform both the training and testing phases of HDC-based outlier detection in memory since SRAM has a much lower writing cost than NVMs. (2) Easier prototyping and fabrication, as CMOS is readily available as opposed to NVMs. (3) Computation in the digital domain which reduces the need for sophisticated peripherals such as ADCs, DACs, and current-based programming circuits. (4) Our general-purpose CiM architecture has the ability to easily accommodate changes in the algorithm (as long as implementing them only requires the same key operations of HDC, which are binding, bundling, and permutation). (5) Prior research on system-level integration and compiler support for CiM architectures, such as [23], could be readily leveraged to bolster the integration of our CiM architecture into a broader computing stack as it can support all the operations realized by **IM-ODHD**.

**IM-ODHD** is depicted in Fig. 2. The design contains $P \times Q$ subarrays, each of which acts as a processing element (PE) in the CiM architecture. Each PE contains $M \times N$ SRAM cells. The tile-styled architecture enables high throughput for the HDC operations (binding, bundling, permutation) due to parallel computation across the different subarrays. The elements of the mat-level design (depicted in Fig. 2(a)) are explained in Section IV-A1. The subarray design with its storage and computing capabilities is discussed in detail in Section IV-A2.

*1) Mat Design:* Our CiM architecture implements the following new elements — decoders, registers, and buses — which enable computation at the mat level. Below, we describe each component of the architecture in detail.

***Decoders:*** Decoders orchestrate data access and facilitate data movement across the different PEs in our **IM-ODHD** fabric. For instance, an example of PE *A* and PE *B* is given in Fig. 2(a) by the tiles colored green and blue, respectively). PE *A* and PE *B* can be accessed concurrently using two pairs of decoders. Two $(log_2P + log_2Q)$-bit addresses are used to activate each of the PEs *A* and *B*. To access the PE *A*, we divide its $(log_2P + log_2Q)$-bit address into two parts; the $log_2P$ most significant bits of the address are used as the input to the row decoder (**Mat-level Row Decoder (PE *A*)** in Fig. 2(a)), and the $log_2Q$ least significant bits of the address are used as the input to the column decoder (**Mat-level Column Decoder (PE *B*)** in Fig. 2(a)). Analogously, when accessing the PE *B*, the $log_2P$ bits of its address are used as the input to the **Mat-level Row Decoder (PE *B*)** in Fig. 2(a), while the $log_2Q$ bits of the address are used as the input to the **Mat-level Column Decoder (PE *B*)** in Fig. 2(a).

***Registers:*** After the decoders select PE *A* and PE *B*, the data from the output of each PE is transferred to its corresponding register, either register *A* or register *B* (as illustrated in Fig. 2(a)). The data traffic between each PE and the registers *A* and *B* is managed via two dedicated buses, which will be elaborated upon in the subsequent paragraph. Once the data has been stored in either register *A* or *B*, it can be rerouted back to any PE through a reverse pathway, which is leveraged by the permutation operations are used in the encoding phase of **ODHD**. Section IV-B provides details about performing this step with **IM-ODHD**.

***Buses A and B:*** Our CiM design employs dedicated buses *A* and *B* to support (1) data movement from/to the PEs to/from registers *A* and *B*, and (2) the setup of a bit shift amount and direction for each PE so reads, divisions, and multiplications by powers-of-two are possible with our **IM-ODHD** fabric. To achieve (1), our proposed CiM architecture has two separate sets of $N$-bit wide buses *A* and *B* that connect each PE to the registers *A* and *B*. The width of the buses is chosen as $N$ so it matches the dimensions of an individual $M \times N$ PE. Furthermore, two pairs of selector lines come out of the row/column decoders and spread through the $P \times Q$ PEs on the mat (see green and blue wiring in Fig. 2(a)). These lines are used to select the PE *A* and the PE *B* for data transfer. Note that only two tiles, i.e., PE *A* and the PE *B*, can be selected at a given time through each pair of decoders, which avoids data conflicts on buses *A* and *B*. For (2), our proposed CiM architecture implements a $3 \times P \times Q$-bit wide bus on which the bit shift amounts used at each PE can be set up individually (more details about bit shifts with our CiM architecture can be found in Section IV-A2).

*2) Subarray Design:* Subarrays are the fundamental PEs of our design with their merged storage and processing capabilities. Fig. 2(b) illustrates our SRAM-based processing element (PE), which utilizes 6T-SRAM memory cells, word line decoders X and Y, customized sense amplifiers, write and copy drivers, and logarithmic bit shifters. These components, akin to [18], [19], [20], are crucial for facilitating the necessary bundling, binding, and permutation operations required by **ODHD**. For an in-depth understanding of the role each component plays within the PE, readers can refer to Section II-B2.

Importantly, besides building on these established PE structures, our work introduces near-memory computing (NMC) circuits at the mat level, such as the buses and auxiliary registers managed by decoders (described in Section IV-A1). The introduced NMC circuits enhance our design's ability to carry out permutations — a feature uniquely tailored to **ODHD**'s encoding phase that represents a departure from previous in-SRAM computing solutions. The introduction of NMC elements in our CiM architecture sets our work apart, as existing in-SRAM architectures do not address the challenge of data movement between CiM PEs.

### B. Hardware/Software (HW/SW) Codesign

This section explains the efficient mapping of the steps of the **ODHD** algorithm (Section III) to the CiM architecture of **IM-ODHD** (presented in Section IV-A). The mapping process adopts the HW/SW codesign principle to adjust the **ODHD** algorithm to better utilize the capabilities of **IM-ODHD**. HW/SW codesign enforced in CiM architecture can significantly increase the performance of **ODHD** while having high accuracy, F1-score, and AUC, as evaluated and discussed in Section V.

*1) Seed HVs Generation in IM-ODHD:* The initial step of creating the seed HVs involves generating them externally using random bit flips. However, once $k$ seed HVs, each with $D$ dimensions, are produced, they get distributed across the $P \times Q$ PEs of the **IM-ODHD** fabric. The PEs have a size of $M \times N$, where $M$ is the number of rows, and $N$ is the number of columns. When a $D$-dimensional seed HV is distributed across the PEs, its elements are indexed to the row $i$ of each PE, where $i \epsilon [1, M]$. The storage of all the elements within each seed HV spreads across $D/N$ PEs of the **IM-ODHD** fabric. Hence, to store $k$ seed HVs, the size of the GPCiM architecture needed is $k \times D$, i.e. $P \times Q \times M \times N \geq k \times D$. This $k \times D$ segment of the **IM-ODHD** fabric is designated for our seed HV storage and remains unaltered throughout the computation.

*2) Encoding in IM-ODHD:* Once the $k$ seed HVs are written to the **IM-ODHD** fabric, the next step is to encode a given feature vector into an HV. The encoding step involves applying **permutation** and **bundling** operations. **Permutation on IM-ODHD** is implemented with circular shifts. The process of performing a circular shift is implemented in two rounds, as follows.

**Round 1 (R1):** Assume a $D$-dimensional seed HV is mapped to the $i^{th}$ row of $D/N$ PEs. We simultaneously access $i^{th}$ row of the $D/N$ PEs holding the HV, and refer to the PEs as the *destination* and *source* PEs, in an alternate fashion. The $i^{th}$ row data at the source PE undergoes a bitwise AND operation with a pre-stored mask filled with 1's at the $m$ least significant bit positions and 0's at the remaining $N - m$ positions (recall from Section III that $m$ corresponds to the number of bits for the circular shift). The resulting value is shifted left by $N - m$ bits using a logarithmic bit shifter and temporarily stored in register A[1]. At the same time, the data on the destination PE is shifted right by $m$ bits and saved in a spare row in the same subarray using the copy drivers. The value from register A is moved to a second spare row in the destination PE, and an OR operation is performed between the values in these two spare rows to produce a circular right-shifted value, which is stored in a third spare $j^{th}$ row in the destination PE. Note that the original data of the source and destination PEs remain intact in $i^{th}$ row during the permutation.

**Round 2 (R2):** In round 2, the former source PE becomes the new destination PE, and the process described for round 1 repeats until all PEs holding the HV have been used as destination PEs once. Afterwards, all the permuted values stored in spare $j^{th}$ rows are copied to $i^{th}$ row of bundle segment of **IM-ODHD** to store the newly encoded HV.

In Fig. 3, we depict an example for the steps involved in the two-round permutation with IM-ODHD. The example performs a 2-bit circular shift (amount of 2 bits, to the right) on the string 'ABCDEFGHIJKLMNOP', which results in 'OPABCDEFGHIJKLMN'. The string is grouped into substrings of 4 characters and stored in four PEs, labeled as source (src) and destination (dest) PEs. A step-by-step explanation of the permutation with **IM-ODHD** is below:

**R1-step (a), Fig. 3(a):** Initially, the substrings in the source PEs are subjected to an in-memory AND operation with a '0011' mask.

**R1-step (b), Fig. 3(b):** The masked substrings from Step (a) undergo a left shift, and the results get stored in registers A and B, placed near the CiM PEs.

**R1-step (c), Fig. 3(c):** Simultaneous to step (a), the substrings in the destination PEs are subjected to an in-memory AND operation with a '1100' mask.

**R1-step (d), Fig. 3(d):** Parallel to step (b), the masked substrings from Step (c) undergo a right shift, and are stored in the 1st spare row in the destination tiles.

**R1-step (e), Fig. 3(e):** Substrings in registers A and B got moved to a 2nd spare row in the destination PEs.

**R1-step (f), Fig. 3(f):** The results of an in-memory OR between the contents of the 1st and 2nd spare rows get stored in the 3rd spare row of the destination PEs, marking the end of the first round of permutation.

**R2, Fig. 3(g):** Steps (a) through (f) of R1 repeat, with source PEs becoming destination PEs and vice-versa.

Going through the steps (a) through (f) of round 1, and round 2, permutes the original string 'ABCDEFGHIJKLMOP', resulting in 'OPABCDEFGHIJKLMN'.

[1]**IM-ODHD** can perform two of such operations by using the register B to store the results, simultaneously to register A.

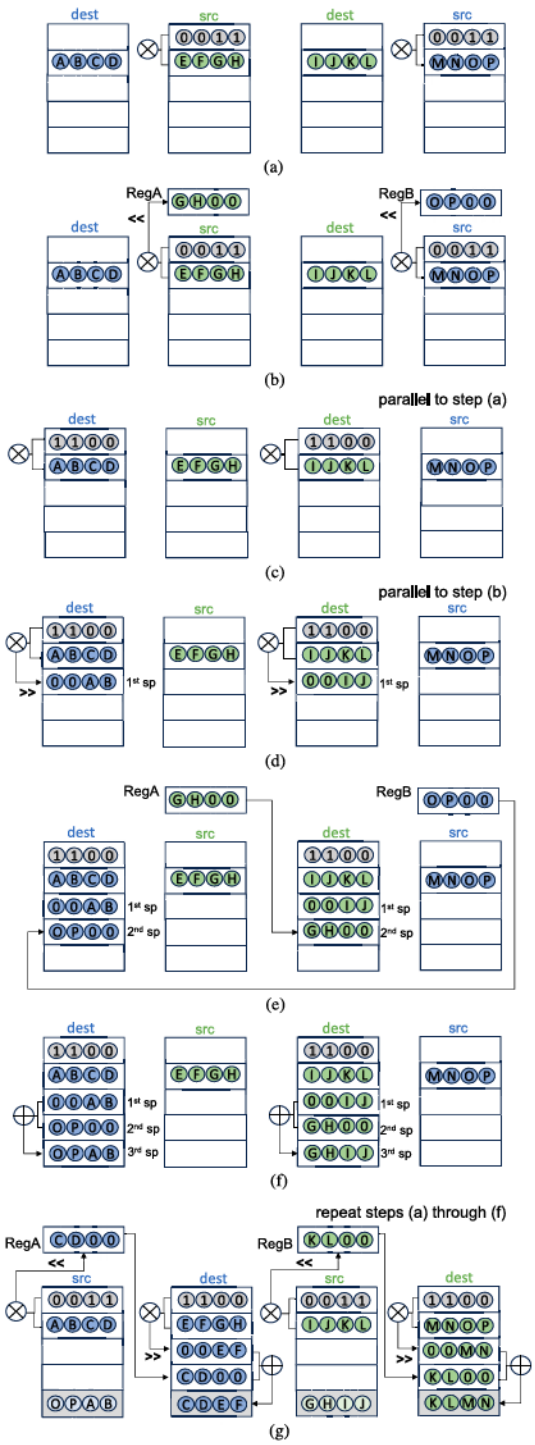



Fig. 3. An example for the permutation with our CiM architecture; (a-f) corresponds to the steps of round 1; (g) depicts round 2.

**Bundling on IM-ODHD** is implemented through the in-memory addition of the $m$ permuted HVs with the CSA. Before the in-memory addition, the permuted HVs are positioned in the **IM-ODHD** fabric so that the HV elements of the same column map to the same PE (but in different rows). For this, at first we copy the $1^{st}$ un-permuted feature HV $\overrightarrow{s_{f_0}}$ from the seed HV segment to the $1^{st}$ spare row of **IM-ODHD** fabric. The permuted HV for the second element of the original feature vector is duplicated to the next spare row. Once the positioning is done, the two HVs are added at a time with the CSA and the intermediate result is overwritten to the $1^{st}$ row. In the next round, the content of this row is added to the next permuted HV, until the bundling of all HVs is concluded. The result generates one encoded HV i.e. feature vector $\overrightarrow{F_n}$ per training dataset. To get the encoded feature HV of the next training sample we copy the $1^{st}$ un-permuted feature HV $\overrightarrow{s_{f_0}}$ from the seed HV segment to the $2^{nd}$ row of bundle segment of **IM-ODHD** fabric and repeat the operation for all of the training dataset in the consecutive rows.

*3) Training in **IM-ODHD**:* Our CiM architecture generates the one-class HV ($\overrightarrow{H_{OC}}$) of the entire training set, i.e., all inlier samples, leveraging the bundling operation exactly as described in Section IV-B2.

*4) HW/SW Codesign for Threshold Calculation:* The threshold calculation in Eq. 5 uses mean and standard deviation, which requires division and a square root operation, which are not well supported by our proposed CiM architecture. Therefore, to make threshold calculation less computationally expensive to implement with our CiM architecture, we carry out three modifications to the algorithm proposed in Section III. Namely, when running **ODHD** on **IM-ODHD**, we (1) realize division with bit-shifts (enabled by logarithmic bit shifters), (2) modify the cosine similarity calculation, and (3) replace standard deviation with a more CiM-friendly mean absolute deviation (MAD) metric.

For (1), most CiM architectures (including ours) are not designed to efficiently support the division. In **IM-ODHD**, division with our CiM hardware is approximated by shifting a binary value by $m$ bits to the right, which divides the value by $2^m$ and rounds down. The logarithmic bit shifters in **IM-ODHD** support a shift amount of 0-3 bits to the left or right. Therefore, divisions up to $2^3$ are possible, which are controlled by the shift mask (see Fig. 2). Moreover, larger shift amounts (for larger divisors) are supported with multiple rounds of bit shifting. Since we need the division operation for calculating $\mu(S)$, we increase the training set such that the number of training samples equals a power of two value. This is done by copying random samples from the original training set without replacement. Doing so may increase the training time since the encoding phase has more samples to extract the information from. The impact of this is reflected in the training time presented in Section V-D.

In the case of (2), while cosine similarity is used in Section III and in [11], our version of **IM-ODHD** uses only the numerator part of the cosine distance (see Eq. 1) to make the architecture more amenable to CiM by eliminating square and square root operations [24]. This is basically the binding operation or the dot product of two HVs, i.e., the one-class HV $\overrightarrow{H_{OC}}$ and the training HVs, which generates the similarity array $S$, followed by a sum (as shown in Eq. 7). This sum operation, essentially a pop-count (counting the number of 1s in a vector), is executed across several cycles. This operation hinges on accumulating partial sums using in-memory adders and bit shifters, key components of the subarrays in **IM-ODHD**.

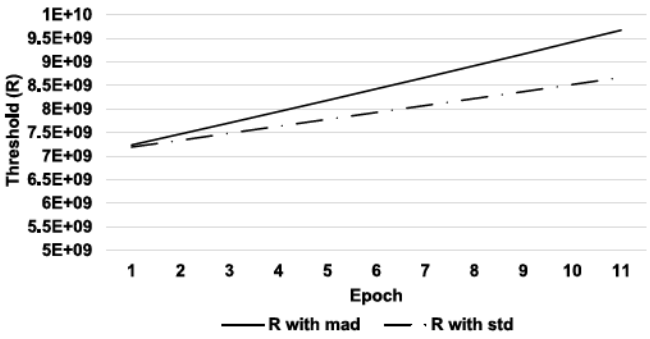



Fig. 4. Threshold trend on MNIST dataset with both standard deviation and mean absolute deviation metrics.

Our evaluation incorporates this multi-cycle approach, which consists of the accumulation of these partial sums with the mentioned circuits.

$$\delta(\overrightarrow{H_x}, \overrightarrow{H_y}) = \overrightarrow{H_x} \cdot \overrightarrow{H_y} = \sum_{i=1}^{d} h_{xi} \cdot h_{yi} \tag{7}$$

Regarding (3), mean absolute deviation (MAD) is defined as the average absolute deviation of a set of values from their mean. MAD is calculated by finding the absolute difference between each data point and the mean (calculated in the previous step), summing these differences, and then dividing by the number of data points. All these operations (modular subtraction, addition, and division with bit shifts) are promptly supported by the components of our CiM architecture. Subtraction, for instance, can be performed as a 2's complement subtraction where we first negate the subtrahend with a NOT operation, perform a local write of the result to the same subarray with a copy driver, and then finally perform an in-memory addition setting the carry of the first bit to 1.

Both standard deviation and MAD are measures of how much the data points in a set deviate from the mean. The absolute value function used in MAD treats positive and negative deviations equally, making it more robust to outliers and emphasizing extreme values in the dataset. As a result, MAD focuses more on the extreme values in the dataset than standard deviation does. Fig. 4 shows the epoch-wise threshold increment for the MNIST dataset using both methods described in Section III-D and Section IV-B4. We observe a higher threshold for MAD, with a difference that leads to the need for more epochs. Using the described methods, the mean value $\mu(S)$ and the mean absolute deviation $MAD(S)$ are computed for each similarity value in the array $S$ to calculate the threshold $R$. The small modifications to the threshold estimation approach described in Section III-D are reflected in Eq. 8.

$$R = \mu(S) + 2 * MAD(S) \tag{8}$$

*5) HW/SW Codesign for Fine-Tuning:* As detailed in Section III, fine-tuning is used to ensure that all training HVs will be correctly identified as inliers. During each fine-tuning epoch, we use all the training samples, previously encoded in the encoding phase of **IM-ODHD**, and for each individual training sample, we calculate the similarity between its HV and the one-class HV $\overrightarrow{H_{OC}}$ using Eq. 7. All the misclassified inliers are then updated into the one-class HV $\overrightarrow{H_{OC}}$ exploiting the in-memory addition with the CSA as described in Section II-B2. Once fine-tuning is accomplished, we no longer need to store the encoded training samples and only store the one-class HV $\overrightarrow{H_{OC}}$ along with the seed HVs that are used in Section IV-B6.

*6) Outlier Detection in IM-ODHD:* Once we have trained the $\overrightarrow{H_{OC}}$ and established the threshold $R$ using the described CiM-friendly method, we deploy **ODHD** to detect outliers in unseen samples without knowledge of their labels. During the outlier detection phase (i.e. the inference/test phase), we encode the testing sample $q$ into a query HV using the same encoding process described in Section IV-B2 and the same seed HVs. We then calculate the similarity between the query HV and the one-class $\overrightarrow{H_{OC}}$ using the method described in Section IV-B5. If the similarity is lower than the predetermined threshold, sample $q$ is classified as an outlier.

## V. Evaluation

In this section, we evaluate the performance of **ODHD** on six datasets and compare the CiM, CPU, and GPU-based implementations of **ODHD** with four baseline methods.

### A. Experimental Setup

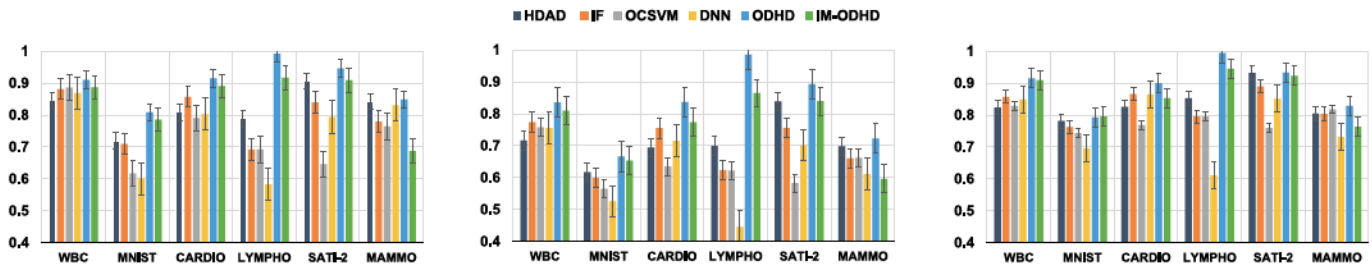
Herewith, we discuss the experimental setup for our software and hardware-level evaluations.

*1) Software Evaluation:* We evaluate the performance of **ODHD** on six datasets selected from the Outlier Detection Datasets (ODDS) Library [12] spanning multiple application domains such as medical diagnosis and wireless communication. These datasets are Wisconsin-Breast Cancer (Diagnostics) dataset (WBC), Mammography (MAMMO), MNIST, Cardiotocography (CARDIO), lymphography (LYMPHO), and Landsat Satellite (SATI2). These datasets are widely used as benchmarks in existing outlier detection studies [25], [26]. Each dataset contains a certain number of outliers specified by the ODDS library, e.g., the WBC dataset has 21 outliers. The testing dataset is mixed inliers and outliers, e.g., 25

We repeat the experiments independently 10 times and report the average performance. We also present error bars as shown in Fig. 5 to illustrate the performance variations due to the randomness in different learning methods.

We compare our modified **ODHD** implemented on **IM-ODHD** discussed in Section IV-B with the original **ODHD** algorithm proposed in [11] and discussed in Section III, as well as with the following four baseline outlier detection methods:

- **Autoencoder**: Autoencoder is an emerging unsupervised learning outlier detection approach based on a neural network. In this paper, we use the same autoencoder architecture as [5].
- **Isolation Forest**: Isolation Forest is an ensemble model of isolation trees, which uses the path length of each sample to detect outliers. In this paper, we establish an isolation forest model using the same configuration as [4].




(a) ACC (b) F1 (c) AUC

Fig. 5. Comparison between **ODHD**, **IM-ODHD** and four baseline methods based on three metrics, ACC, F1 and ROC.

- **OCSVM**: OCSVM attempts to separate outliers from the inliers with the maximum margin. We have a grid search for an appropriate set of hyper-parameters such as kernel functions and the value of gamma to fine-tune the OCSVM model following [27].
- **HDAD**: HDAD follows similar principles of autoencoder; it first "reconstruct" the input samples and then detects anomalies based on reconstruction error. We use the same architecture of [22].

We implement **ODHD** and the four baseline methods in Python and perform our experiments on a desktop with an i7-7700 CPU, 12 GB RAM, and an NVIDIA P1000 GPU with 4 GB onboard memory. We implement the GPU version ODHD based on Pytorch and use the HWiNFO tool [28] to measure energy consumption. HWiNFO is a commercial tool for monitoring hardware circumstances and has been utilized in previous work [29], [30].

Unlike traditional DNN operations such as the Conv2D layer, the GPU version of ODHD does not have a specialized data flow or CUDA optimization. In the GPU implementation, the most time-consuming part is data transfer between the CPU and GPU memory. Since the HV are high-dimension vectors, the GPU acceleration can be slowed down by the data transfer. According to our experimental results, the GPU version of ODHD provides ∼2–2.5× time efficiency compared with the CPU version of ODHD, which is consistent with the results presented in torchHD [31].

To comprehensively assess the algorithm-level performance of **ODHD**, we use three metrics: accuracy (ACC), F1 score (F1), and Area under ROC curve ROC-AUC (AUC). Note that while accuracy is widely used and easy to understand, an outlier detection dataset may be significantly imbalanced. Hence, accuracy may not precisely reveal the performance of outlier detectors. Therefore, we also use ROC-AUC, which is widely used for outlier detection as it can accurately represent the tradeoff between true positive and false positive [32]. Meanwhile, the F1 score is also a widely-used metric in binary classification which can comprehensively indicate the tradeoff between precision and recall [33].

*2) Hardware Evaluation:* Besides the accuracy, F1, and AUC, which are used to evaluate the algorithm-level performance of **ODHD**, we measure the runtime and energy of **ODHD** based on different platforms, i.e., CPU, GPU, and CiM, to capture performance tradeoffs. For the CPU version of **ODHD**, we measure the training and testing runtime of the application running on a desktop with an i7-7700 CPU, 12 GB RAM, and an NVIDIA P1000 GPU with 4 GB on-board memory.

TABLE I
PARAMETERS USED IN OUR EVALUATION OF THE CIM MAT

| Parameter | P | Q | M | N |
|---|---|---|---|---|
| **PE (L): Large PE** | 16 | 16 | 1024 | 1024 |
| **PE (M): Medium PE** | 32 | 32 | 512 | 512 |
| **PE (S): Small PE** | 64 | 64 | 256 | 256 |

For the CiM implementation of **ODHD**, which we refer to as **IM-ODHD**, we simulated the SRAM-based CiM mat of Fig. 2 using Destiny [34], a tool for modeling emerging 2D and 3D NVM and SRAM caches, which was extended with the customized peripheral circuits employed by **IM-ODHD** (i.e., customized sense amplifiers, logarithmic bit shifters, an extra wordline decoder, and copy drivers). In our evaluation, we employ the CMOS Predictive Technology Model (PTM) from [35], specifically designed for a 45nm technology node to simulate CiM circuits. Furthermore, our evaluation of **IM-ODHD** accounted for all the components shown in Fig. 2 in addition to those within each subarray. The registers, mat-level decoders, and communication network were implemented using Verilog and synthesized with Cadence Encounter RTL Compiler v14.10, using the NanGate 45nm open-cell library [36].

To accommodate a wide range of datasets with our SRAM-based CiM architecture, we must carefully select values for P, Q, M, and N. We conduct a design space exploration for the **IM-ODHD** mat parameters, as outlined in Table I. Table II summarizes the latency and energy consumption of various in-memory operations across the three simulated design configurations from Table I.

Notably, PE (S) exhibits shorter latency, along with reduced energy consumption per PE. However, PE (S) also results in an increased number of dedicated PEs, which necessitates an expanded global address line and a larger amount of memory peripherals, leading to a potential area disadvantage with respect to PE (M) and PE (L). In contrast, PE (L) results in less peripheral circuitry and it is more area efficient, at the same time providing satisfactory latency and energy. To establish a lower-bound for the performance of **IM-ODHD**, we employ PE (L) throughout our latency and energy evaluation (Section V-D).

TABLE II
LATENCY (NS) AND ENERGY (NJ) FOR IM OPERATIONS, WITH RESPECT TO THE ARCHITECTURES DEFINED IN TABLE I

| Operation | Latency (ns) | | | Energy (nJ) | | |
|---|---|---|---|---|---|---|
| | PE (L) | PE (M) | PE (S) | PE (L) | PE (M) | PE (S) |
| Read/NOT | 5.24 | 2.64 | 1.42 | 17.36 | 5.51 | 1.66 |
| AND/OR | 5.28 | 2.68 | 1.48 | 18.44 | 18.40 | 2.50 |
| PointwiseMult. | 5.28 | 2.68 | 1.48 | 18.44 | 18.40 | 2.50 |
| Write | 5.08 | 2.46 | 1.26 | 14.58 | 6.78 | 0.96 |
| Add | 12.87 | 10.20 | 9.04 | 19.97 | 96.30 | 47.30 |
| Sub | 17.96 | 12.70 | 10.30 | 21.43 | 103.08 | 48.21 |
| Shift | 5.24 | 2.64 | 1.42 | 17.36 | 5.51 | 1.66 |
| Permut* | 36.13 | 17.80 | 9.40 | 93.58 | 69.50 | 10.50 |

*For permutation, ∼56.2% of the time (∼37.7% of the energy) is spent on operations between the PEs and the registers, while the rest of the time (energy) is spent on operations performed within the PEs.

### B. Performance Evaluation for ODHD

As shown in Fig. 5, we compare the performance of **ODHD** and **IM-ODHD** with the baseline methods on six datasets for the three metrics *with error bar*. **IM-ODHD** performs better than the baseline models in all but one case (as will be discussed in Section V-C). On the other hand, **ODHD** is able to consistently outperform the four baseline methods **on every dataset for every metric**.

First, for F1, the average F1 of **ODHD** is 82.3% on all datasets, representing an improvement of 18.5% over OCSVM, 12.8% over isolation forest, 15.8% over HDAD and 19.8% over autoencoder. For AUC, the average AUC of **ODHD** is 89.4%, representing an improvement of 10.9% over OCSVM, 6.5% over isolation forest, 6.8% over HDAD, and 12.7% over autoencoder.

Second, while **ODHD** has a certain level of fluctuation (error bars) in different runs (just like all the other models), we can observe that even the low end of **ODHD** is higher than the high end of any baseline method, representing the robustness of the performance of **ODHD**.

Third, while the performance of different methods varies with different datasets, **ODHD** shows better stability compared to other methods. For example, for ACC, the lowest ACC of **ODHD** is over 80%, while the lowest ACC are about 60%, 70%, 60% and 70% for OCSVM, Isolation Forest, Autoencoder and HDAD, respectively. A similar phenomenon can also be seen in F1 and AUC.

Last but not least, in certain datasets, e.g., LYMPHO, all baseline methods significantly underperform while **ODHD** maintains a high accuracy close to 100%. The reason is possibly related to the fact that the lymphography data are relatively small so that the baseline methods cannot converge to a proper point; however, **ODHD** is able to learn useful information even from a small amount of data. Similar advantages of HDC have been observed in various supervised classification studies for biomedical datasets that are often small [37].

### C. Performance Evaluation for IM-ODHD

From Fig. 5 it is evident that **IM-ODHD** shows modest accuracy loss with respect to **ODHD** due to algorithm-level modification of **ODHD**. To better understand the extent of this accuracy loss, we analyze the average results for all datasets (presented in Fig. 6).

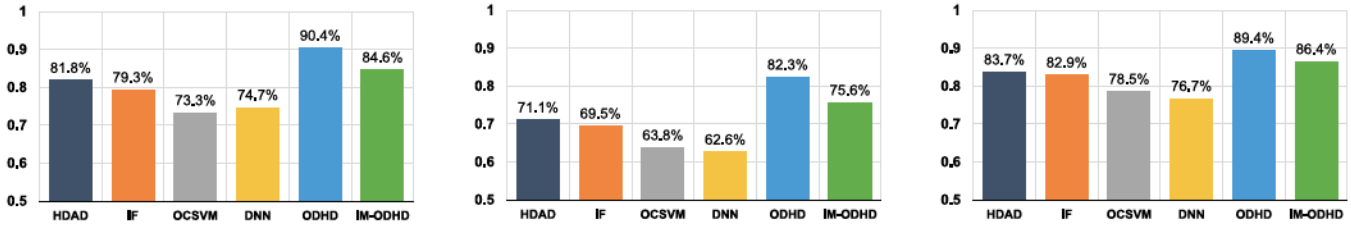
For ACC, the average ACC of **ODHD** is 90.4% on all datasets, representing an improvement of 17.1% over OCSVM, 11.1% over isolation forest, 10.5% over HDAD and 15.7% over autoencoder. The average ACC, F1 score of **IM-ODHD** degrades 6.37%, 8.1% respectively compared to **ODHD**. The most important metric for outlier detection, ROC-AUC degrades only 3.3% i.e., **IM-ODHD** can still satisfactorily represent the tradeoff between true positive and false positive. It is important to highlight that, even though these metrics do not surpass the software level accuracy for **ODHD**, **IM-ODHD** still shows better performance in terms of the average accuracy i.e. 3.4% over HDAD, 6.6% over isolation forest, 15.4% over OCSVM, 13.3% over DNN, the average F1 score i.e. 3.4% over HDAD, 6.6% over isolation forest, 15.4% over OCSVM, 13.3% over DNN and the average ROC i.e. 3.4% over HDAD, 6.6% over isolation forest, 15.4% over OCSVM, 13.3% over DNN.

We observe that the performance of **IM-ODHD** is heavily dependent on how well the high variability feature HVs capture the diversity of each sample during the training stage. The results show that **IM-ODHD** does not perform as well with the mammography dataset as it does on other datasets, achieving an accuracy of 68.7% and an F1-score of 59.6%, which is lower than other existing models. However, **IM-ODHD** exhibits a 3.4% improvement in the ROC-AUC metric compared to the baseline model DNN. Therefore, although **IM-ODHD** can achieve high accuracy with small datasets like lympho, its performance is highly dependent on the ability of the HVs to interpret the data from the features. With an increase in the number of features in the dataset, the accuracy of **IM-ODHD** approaches the software level (i.e., **ODHD**) accuracy. For instance, with the MNIST dataset, which has the highest number of features among all the six datasets, the accuracy of **IM-ODHD** is only 1% less accurate than **ODHD**'s accuracy.

### D. Latency and Energy Evaluation

We also evaluated the execution time of outlier detection with the different models and datasets. Table III and Table IV present a breakdown of latency/energy with **IM-ODHD** during the different phases of training and testing, respectively. For instance, the **encoding** of samples into HVs is required for both training and testing phases and requires permutation to be performed on the seed HVs, followed by the bundling operation. While bundling can be performed entirely within the PEs in the **IM-ODHD** architecture, permutation requires transfers to/from registers A and B placed at the mat level in our CiM architecture. During training (the most expensive portion of **ODHD**), on average across all datasets, encoding accounts for ∼88%/∼68.55% of the latency/energy. Communication from PEs to registers A and B, and vice-versa, dominate the costs of encoding, accounting for ∼58.7%/∼63.2% of its latency/energy.

Since the latency/energy of encoding during testing is still (∼99.31%/∼98%, on average, with respect to the total testing




(a) ACC (b) F1 (c) AUC

Fig. 6. Average performance of different models over six datasets.

TABLE III
LATENCY/ENERGY BREAKDOWN OF TRAINING WITH **IM-ODHD** USING PE (L)

| Dataset | Encoding: Permutation IM Ops w. PEs ($\mu s/\mu J$) | PE-to-REG Comm. ($\mu s/\mu J$) | REG-to-PE Comm. ($\mu s/\mu J$) | Enc: Bundling IM Ops w. PEs ($\mu s/\mu J$) | Bndl+Thr+Tun IM Ops w. PEs ($\mu s/\mu J$) | Total ($\mu s/\mu J$) |
|---|---|---|---|---|---|---|
| WBC | 307.6/980.8 | 153.3/474.3 | 75.5/216.5 | 17.2/205.8 | 91.1/1039.6 | 644.8/2917.0 |
| MNIST | 16802.5/53570.3 | 8374.5/25908.5 | 4121.5/11826.9 | 919.2/10973.4 | 1450.8/16548.4 | 31668.5/118827.5 |
| CARDIO | 848.6/2705.6 | 423.0/1308.5 | 208.2/597.3 | 48.3/576.1 | 363.0/4140.0 | 1890.9/9327.5 |
| LYMPHO | 45.1/143.7 | 22.5/69.5 | 11.1/31.7 | 2.6/30.9 | 22.8/260.0 | 104.0/535.8 |
| SATI | 5940.3/18939.0 | 2960.7/9159.6 | 1457.1/4181.2 | 330.9/3950.4 | 1459.1/16647.6 | 12148.1/52877.8 |
| MAMMO | 1697.2/5411.1 | 845.9/2617.0 | 416.3/1194.6 | 110.3/1316.8 | 2903.8/33123.7 | 5973.5/43663.3 |

TABLE IV
LATENCY/ENERGY BREAKDOWN OF TESTING WITH **IM-ODHD** USING PE (L)

| Dataset | Encoding: Permutation IM Ops w. PEs ($\mu s/\mu J$) | PE-to-REG Comm. ($\mu s/\mu J$) | REG-to-PE Comm. ($\mu s/\mu J$) | Enc: Bundling IM Ops w. PEs ($\mu s/\mu J$) | Outlier Detection IM Ops w. PEs ($\mu s/\mu J$) | Total ($\mu s/\mu J$) |
|---|---|---|---|---|---|---|
| WBC | 50.7/161.7 | 25.3/78.2 | 12.4/35.7 | 2.8/33.9 | 1.5/18.7 | 92.8/328.1 |
| MNIST | 5743.0/18310.2 | 2862.4/8855.4 | 1408.7/4042.4 | 314.2/3750.7 | 43.6/544.2 | 10371.9/35502.9 |
| CARDIO | 291.7/930.0 | 145.4/449.4 | 71.6/205.3 | 16.6/198.0 | 12.0/136.8 | 537.2/1920.0 |
| LYMPHO | 8.6/27.5 | 4.3/13.3 | 2.1/6.1 | 0.5/5.8 | 0.5/6.2 | 16.1/58.8 |
| SATI | 206.2/657.4 | 102.8/317.9 | 50.6/145.1 | 11.5/137.0 | 4.9/56.0 | 375.9/1313.4 |
| MAMMO | 107.7/343.5 | 53.7/166.1 | 26.4/75.8 | 7.0/83.6 | 17.7/202.1 | 212.5/871.2 |

TABLE V
EXECUTION TIME (MS) OF DIFFERENT OUTLIER DETECTION MODELS OVER SIX DATASETS (TRAINING TIME/TESTING TIME)

| | OCSVM | Isolation Forest | HDAD | ODHD | ODHD(GPU) | **OM-ODHD** |
|---|---|---|---|---|---|---|
| WBC | 3.000/2.000 | 112.0/30.00 | 412.0/198.0 | 399.0/112.0 | 187.0/62.00 | 0.645/0.093 |
| MNIST | 925.0/782.0 | 355.0/198.0 | 20773/25662 | 18024/9631 | 8872/6159 | 31.67/10.372 |
| CARDIO | 31.00/45.00 | 115.0/42.00 | 1134/1248 | 1212/615.0 | 511.0/442.0 | 1.89/0.537 |
| LYMPHO | 1.000/1.000 | 111.0/28.00 | 169.0/53.00 | 119.0/20.00 | 45.00/12.00 | 0.104/0.016 |
| SATI2 | 965.0/84.00 | 211.0/36.00 | 7205/704.0 | 9109/549.0 | 4194/267.0 | 12.148/0.376 |
| MAMMO | 1942/478.0 | 151.0/44.00 | 7474/1129 | 5644/555.0 | 2215/382.0 | 5.974/0.213 |

latency/energy), the cost of outlier detection with **IM-ODHD** is insignificant, due to the need for successive shifts and additions in the implementation of the pop-count operation in the threshold calculation. Communication from PEs to registers A and B, and vice-versa, during encoding, is similar to the training phase, accounting for ~58.7%/~63.1% of the encoding latency/energy.

Table V shows the execution time for training and testing including the GPU implementation of **ODHD** and the CiM-based implementation (**IM-ODHD**), along with other baseline methods. In general, conventional models execute faster outlier inference than HDC-based models on the CPU. With a significant amount of cores and faster data transmission between memory and computing unit, GPU achieves lower

TABLE VI
ENERGY (MJ) COMPARISON **ODHD** VS.**IM-ODHD**

| Dataset | ODHD (GPU) | | **OM-ODHD** | |
|---|---|---|---|---|
| | Training Energy | Testing Energy | Training Energy | Testing Energy |
| WBC | 17.00 | 6.200 | 2.917 | 0.328 |
| MNIST | 2162 | 1337 | 118.828 | 35.503 |
| CARDIO | 50.70 | 42.30 | 9.328 | 1.92 |
| LYMPHO | 5.400 | 3.500 | 0.536 | 0.059 |
| SATI2 | 770.50 | 49.10 | 52.878 | 1.313 |
| MAMMO | 1949 | 358.0 | 43.663 | 0.871 |

execution time. However, the HDC-based model still takes a longer time to train and infer than conventional models, e.g., OCSVM and isolation forest.

Our proposed **IM-ODHD** significantly accelerates both the training and testing phases of outlier detection. According to Table V, **IM-ODHD** shows on average 331.5× speedup in training and 889× speedup in inference than **ODHD** running on GPU (the fastest implementation). The training time of **IM-ODHD** is slightly large due to the working principle of **IM-ODHD** that is amiable with CiM architecture, yet shows extensively superior performance since it is minimal compared to other baseline models for outlier detection. It is challenging to train the MNIST dataset because the model must learn a representation of the input images that is resilient to changes in writing style, stroke thickness, and other elements that can impact how the digits appear. **IM-ODHD** can completely learn this dataset in 31.67ms, with an inference time of 10.37 ms whereas isolation forest takes 355ms/198ms to train/test on the same dataset. Small datasets like Lympho can be learned in 104$\mu$s and infer any outlier in 16.1$\mu$s using **IM-ODHD**.

Last, energy results are reported in Table VI for the training and testing phases of **IM-ODHD**. Due to highly parallel calculation in **IM-ODHD** fabric, the energy consumption, which factors in both power and latency, is advantageous compared to the GPU-based implementation of **ODHD**. On average, energy improvement for **IM-ODHD** is at 14.0× / 36.9× for the training/testing phase.

## VI. RELATED WORK

In this section, we review related work on models for outlier detection and hardware accelerators for HDC.

### A. Models for Outlier Detection

Outlier detection has been a heavily researched topic with various statistical and machine learning methods proposed. One widely-used outlier detection method is the Exemplar-Based Gaussian Mixture Model (GMM) proposed by Yang et al. [38], which utilizes a globally optimal expectation maximization (EM) algorithm to fit the GMM to the given dataset. Tang et al. [1] further improved this method by combining GMM with locality-preserving projections. Another approach uses linear regression, such as the method proposed by Satman et al. [39], which detects outliers based on a non-interactive covariance matrix and concentration steps applied in the least trimmed square estimation. However, despite their mathematical robustness, statistical methods' assumptions and dependence on a particular distribution model may limit their practical use. **ODHD** provides a novel approach to outlier detection that does not rely on specific distributional assumptions, making it a promising alternative to existing methods.

Three widely used machine learning-based outlier detection methods are OCSVM, isolation forest, and autoencoder. OCSVM separates outliers from inliers by maximizing the margin and detects samples outside the estimated region as outliers [3]. In isolation forest, outliers are detected by examining the path length, as they are more sensitive to isolation and have a relatively short traversal path length [4]. Autoencoder, a neural network-based method, consists of an encoding network and a decoding network. The encoder maps input samples to a low-dimensional feature space, while the decoder reconstructs the sample from the encoded feature. Autoencoder is trained to minimize the reconstruction error and preserve information relevant to normal instances. Outliers, which diverge from the majority of training samples, are hardly reconstructed and lead to a high reconstruction error. Thus, the outliers can be detected by examining the reconstruction error [5]. Despite the popularity of these methods, they rely on different assumptions and may not perform well in various applications.

In recent years, several methods have been proposed for anomaly detection using HDC. One such method, HDAD [22], adopts an autoencoder-like approach to reconstruct the input samples and detect anomalies based on reconstruction error. However, this method requires tedious encoding and decoding processes, making the detection process cumbersome. In contrast, **ODHD** proposes a one-class HDC approach for outlier detection, which is fundamentally different from HDAD. We evaluate the performance of **ODHD** against four baseline methods, namely OCSVM, isolation forest, autoencoder, and HDAD, and provide comprehensive comparison results.

### B. CiM Accelerators for HDC

HDC with its inherent memory-centric operations motivates to implement it in CiM since data movement reduction can be achieved by HV computations fully in memory. Nevertheless, recent research on DRAM-based CiM designs is tailored

to parallel Boolean bitwise operations and often lacks comprehensive support for all operations integral to ODHD. For example, AMBIT [40], with triple-row activation can execute bitwise majority function but misses native shift operation support essential for HDC encoding. DRISA [41] allows for shift operations within subarrays at the cost of area overhead with multiple microarchitectures for data movement making them inadequate for host memory. DRAM-based CiM architectures, while offering increased computational speed, entail a substantial overhead in terms of processing time, usually requiring several hundred clock cycles for operations involving inputs exceeding three bits. While this approach may be well-suited for tasks like image classification, it may not be a viable choice when designing the architecture for applications involving HVs of 10,000 dimensions. CiM architectures based on lookup tables (LUTs) within DRAM enable fast operations while preserving application level accuracy (e.g., [42], [43]). However, the use of LUT-based CiM in DRAM may face challenges in managing the size and volume of LUTs required for performing operations in HDC, since the Hypervectors (HV) involved in the computations have thousands of dimensions (10,000+), which requires further investigation.

The outlined issues with DRAM motivate the search for CiM architectures based on non-volatile memories (NVMs) and CMOS-based SRAMs (our work), which could support more intricate operations. For the former, Imani et al. [44] proposed SearcHD, which utilizes the analog properties of ReRAM-based in-memory computing (IMC) arrays to employ HD blocks in memory with a fully binarized computing algorithm. However, the energy and time required to program the MAJ IMC array from the XOR IMC array severely limit their ability to be used efficiently. In our work, we distribute each bipolar HV of D dimensions across the readily available technology CMOS-based SRAM in a holistic way reducing data transfer overheads. Leveraging from the digital domain computation without any ADC/DAC or current controlled PEs, computation is fully exerted in memory using elements with smaller hardware footprints. By realizing training and testing phases without using analog operations, our CiM architecture improves the time complexity and energy consumption without trading off reliability, which makes it a good fit for low-power hardware devices, aligned with other proposed architectures [45].

## VII. Conclusion

In this study, we propose **ODHD**, a novel outlier detection algorithm based on hyperdimensional computing (HDC), a non-traditional machine learning paradigm. Additionally, we present **IM-ODHD**, a computing-in-memory (CiM) hardware and software (HW/SW) co-design implementation to enhance latency and energy efficiency. The proposed **ODHD** algorithm leverages a learning structure to generate a one-class hypervector (HV) based on inlier samples. This HV represents the abstract information of all inlier samples, and any testing sample with an HV dissimilar from this HV is identified as an outlier. Both the training and testing phases of **ODHD** can be performed using conventional CPU/GPU hardware or our proposed SRAM-based CiM architecture using HW/SW co-design techniques. We evaluate the performance of **ODHD** on six datasets from different application domains using three metrics – accuracy, F1 score, and ROC-AUC and compare it with several baseline methods, such as OCSVM, isolation forest, and autoencoder. The experimental results show that **ODHD** outperforms all the baseline methods in terms of these three metrics on every dataset for both CPU/GPU and CiM implementations. Moreover, we conduct an extensive design space exploration to demonstrate the tradeoff between delay, energy efficiency, and performance of **ODHD**. We show that **IM-ODHD**, the in-memory computing-based implementation of **ODHD**, outperforms the GPU-based implementation of **ODHD** by at least 331.5×/889× in terms of training/testing latency and on average 14.0×/36.9× in terms of training/testing energy consumption.

## References


[1] X.-m. Tang, R.-x. Yuan, and J. Chen, "Outlier detection in energy disaggregation using subspace learning and Gaussian mixture model," *Int. J. Control Automat.*, vol. 8, no. 8, pp. 161–170, 2015.

[2] L. J. Latecki, A. Lazarevic, and D. Pokrajac, "Outlier detection with kernel density functions," in *Proc. Mach. Learn. Data Mining Pattern Recognit. (MLDM)*, vol. 7, 2007, pp. 61–75.

[3] Y. Li, T. Zhang, Y. Y. Ma, and C. Zhou, "Anomaly detection of user behavior for database security audit based on OCSVM," in *Proc. 3rd Int. Conf. Inf. Sci. Control Eng. (ICISCE)*, Piscataway, NJ, USA: IEEE Press, 2016, pp. 214–219.

[4] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proc. 8th IEEE Int. Conf. Data Mining*, Piscataway, NJ, USA: IEEE Press, 2008, pp. 413–422.

[5] T. He, L. Zhang, F. Kong, and A. Salekin, "Exploring inherent sensor redundancy for automotive anomaly detection," in *Proc. 57th ACM/IEEE Des. Automat. Conf. (DAC)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1–6.

[6] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cogn. Comput.*, vol. 1, pp. 139–159, 2009.

[7] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circuits Syst. Mag.*, vol. 20, no. 2, pp. 30–47, 2nd Quart. 2020.

[8] M. Hersche, E. M. Rella, A. Di Mauro, L. Benini, and A. Rahimi, "Integrating event-based dynamic vision sensors with sparse hyperdimensional computing: A low-power accelerator with online learning capability," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Des.*, 2020, pp. 169–174.

[9] C. Elkan and K. Noto, "Learning classifiers from only positive and unlabeled data," in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2008, pp. 213–220.

[10] D. Ielmini and G. Pedretti, "Device and circuit architectures for in-memory computing," *Adv. Intell. Syst.*, vol. 2, no. 7, 2020, Art. no. 2000040.

[11] R. Wang, X. Jiao, and X. S. Hu, "ODHD: One-class brain-inspired hyperdimensional computing for outlier detection," in *Proc. 59th ACM/IEEE Des. Automat. Conf.*, 2022, pp. 43–48.

[12] S. Rayana, "Outlier detection datasets (ODDS) library." ODDS. Accessed: Feb 1, 2024. [Online]. Available: http://odds.cs.stonybrook.edu

[13] M. Kang, M. S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, 2014, pp. 8326–8330.

[14] X. Yin, K. Ni, D. Reis, S. Datta, M. Niemier, and X. S. Hu, "An ultra-dense 2FeFET TCAM design based on a multi-domain FeFET model," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 66, no. 9, pp. 1577–1581, Sep. 2018.

[15] A. Kazemi et al., "Achieving software-equivalent accuracy for hyperdimensional computing with ferroelectric-based in-memory computing," *Sci. Rep.*, vol. 12, no. 1, 2022, Art. no. 19201.
[16] J. Liu, M. Ma, Z. Zhu, Y. Wang, and H. Yang, "HDC-IM: Hyperdimensional computing in-memory architecture based on RRAM," in *Proc. 26th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, 2019, pp. 450–453.
[17] D. Reis, M. Niemier, and X. S. Hu, "Computing in memory with FeFETs," in *Proc. Int. Symp. Low Power Electron. Des.*, 2018, pp. 1–6.
[18] D. Reis, A. F. Laguna, M. Niemier, and X. S. Hu, "A fast and energy efficient computing-in-memory architecture for few-shot learning applications," in *Proc. Des., Automat. Test Europe Conf. Exhib. (DATE)*, 2020, pp. 127–132.
[19] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb 2017, pp. 481–492.
[20] D. Reis, J. Takeshita, T. Jung, M. Niemier, and X. S. Hu, "Computing-in-memory for performance and energy-efficient homomorphic encryption," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 11, pp. 2300–2313, Nov. 2020.
[21] Y. Kim, M. Imani, and T. S. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proc. 8th Int. Conf. Internet Things*, 2018, pp. 1–6.
[22] R. Wang, F. Kong, H. Sudler, and X. Jiao, "Brief industry paper: HDAD: Hyperdimensional computing-based anomaly detection for automotive sensor attacks," in *Proc. IEEE 27th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 461–464.
[23] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 397–410.
[24] A. Ranjan, S. Jain, J. R. Stevens, D. Das, B. Kaul, and A. Raghunathan, "X-MANN: A crossbar based architecture for memory augmented neural networks," *Proc. 56th ACM/IEEE Des. Automat. Conf. (DAC)*, pp. 1–6, 2019.
[25] A. Zimek, M. Gaudet, R. J. Campello, and J. Sander, "Subsampling for efficient and effective unsupervised outlier detection ensembles," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 428–436.
[26] S. Sathe and C. Aggarwal, "LODES: Local density meets spectral outlier detection," in *Proc. SIAM Int. Conf. Data Mining*. Philadelphia, PA, USA: SIAM, 2016, pp. 171–179.
[27] S. Wang, Q. Liu, E. Zhu, F. Porikli, and J. Yin, "Hyperparameter selection of one-class support vector machine by self-adaptive data shifting," *Pattern Recognit.*, vol. 74, no. C, pp. 198–211, Feb. 2018.
[28] "Free system information monitoring and diagnostics." HWiNFO. Accessed: Feb 1, 2024. [Online]. Available: https://www.hwinfo.com/
[29] J. Mo, J. Gopinath, and B. Reagen, "HAAC: A hardware-software co-design to accelerate garbled circuits," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, 2023, pp. 1–13.
[30] P. Maxwell, D. Niblick, and D. C. Ruiz, "Using side channel information and artificial intelligence for malware detection," in *Proc. IEEE Int. Conf. Artif. Intell. Comput. Appl. (ICAICA)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 408–413.
[31] M. Heddes, I. Nunes, P. Vergés, D. Desai, T. Givargis, and A. Nicolau, "Torchhd: An open-source Python library to support hyperdimensional computing research," 2022, *arXiv:2205.09208*.
[32] Z. Wang, B. Dai, D. Wipf, and J. Zhu, "Further analysis of outlier detection with deep generative models," *Proc. 34th Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2020, pp. 8982–8992.
[33] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes (VOC) challenge," *Int. J. Comput. Vis.*, vol. 88, no. 88, pp. 303–338, Jun. 2010.
[34] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "DESTINY: A tool for modeling emerging 3D NVM and eDRAM caches," in *Proc. Des., Automat. Test Europe Conf. Exhib. (DATE)*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 1543–1546.
[35] W. Zhao and Y. Cao, "Predictive technology model for nano-CMOS design exploration," *ACM J. Emerg. Technol. Comput. Syst. (JETC)*, vol. 3, no. 1, pp. 1-es, 2007.
[36] J. Knudsen, "NanGate 45nm open cell library," CDNLive, EMEA, 2008.
[37] A. Burrello, K. Schindler, L. Benini, and A. Rahimi, "One-shot learning for iEEG seizure detection using end-to-end binary operations: Local binary patterns with hyperdimensional computing," in *Proc. IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–4.
[38] X. Yang, L. J. Latecki, and D. Pokrajac, "Outlier detection with globally optimal exemplar-based GMM," in *Proc. SIAM Int. Conf. Data Mining*, Philadelphia, PA, USA: SIAM, 2009, pp. 145–154.
[39] M. H. Satman, "A new algorithm for detecting outliers in linear regression," *Int. J. Statist. Probability*, vol. 2, no. 3, 2013, Art. no. 101.
[40] V. Seshadri et al., "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 273–287.
[41] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based reconfigurable in-situ accelerator," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 288–301.
[42] Q. Deng, Y. Zhang, M. Zhang, and J. Yang, "LAcc: Exploiting lookup table-based fast and accurate vector multiplication in DRAM-based CNN accelerator," in *Proc. 56th Annu. Des. Automat. Conf.*, 2019, pp. 1–6.
[43] P. R. Sutradhar, M. Connolly, S. Bavikadi, S. M. P. Dinakarrao, M. A. Indovina, and A. Ganguly, "pPIM: A programmable processor-in-memory architecture with precision-scaling for deep learning," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 118–121, Jul.–Dec. 2020.
[44] M. Imani et al., "SearcHD: A memory-centric hyperdimensional computing with stochastic training," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2422–2433, Oct. 2020.
[45] M. Eggimann, A. Rahimi, and L. Benini, "A 5 $\mu$w standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 10, pp. 4116–4128, Oct. 2021.

**Ruixuan Wang** (Graduate Student Member, IEEE) received the M.Sc. degree in computer engineering from New York University, USA, in 2020. He is currently working toward the Ph.D. degree in computer engineering (CpE) with the Department of Electrical and Computer Engineering, Villanova University. His research interests include deep learning, approximate computing, hyperdimensional computing, machine learning security, and robustness.

**Sabrina Hassan Moon** (Student Member, IEEE) received the B.S. degree from Shahjalal University of Science and Technology, Bangladesh. She is currently working toward the Ph.D. degree in computer science and engineering with the University of South Florida. Her research interests include computing in memory, hardware-software co-design for machine learning applications, emerging devices, device characterization, and VLSI. She is a devoted individual committed to promoting women's contributions in academia.

**Xiaobo Sharon Hu** (Fellow, IEEE) received the B.S. degree from Tianjin University, the M.S. degree from the Polytechnic Institute of New York, and the Ph.D. degree from Purdue University. She is a Professor with the University of Notre Dame. Her research interests include energy/reliability-aware system design, circuit and architecture design with emerging technologies, real-time embedded systems, and hardware-software co-design. She received the NSF CAREER Award in 1997, the Best Paper Award from Design Automation Conference in 2001, ACM/IEEE International Symposium on Low Power Electronics and Design in 2018, etc.

**Xun Jiao** (Member, IEEE) received the B.S. degree from Beijing University of Posts and Telecommunications, in 2013, and the Ph.D. degree from UC San Diego, in 2018. He is an Assistant Professor with the ECE Department, Villanova University. He was a Visiting Scientist with Meta/Facebook, and a Visiting Student Researcher with NXP Semiconductors. His research interests include software-hardware codesign, design automation, bio-inspired computing, and machine learning, with a particular focus on designing robust and energy-efficient systems. His research is funded by NSF, NIH, and industry corporations (L3Harris, NVIDIA).

**Dayane Reis** (Senior Member, IEEE) received the B.S. degree from PUC-MG, Brazil, the M.S. degree from the Federal University of Minas Gerais, Brazil, and the Ph.D. degree from the University of Notre Dame. She is an Assistant Professor with the Department of CSE, University of South Florida. Her research interests include the design of circuits and architectures for data-intensive computing. She was one of the Best Paper Award Winners in the ACM/IEEE International Symposium on Low Power Electronics and Design, and a recipient of the Cadence Women in Technology (WIT) Scholarship 2018/2019.