Store-n-Learn: Classification and Clustering with Hyperdimensional Computing across Flash Hierarchy

SARANSH GUPTA, BEHNAM KHALEGHI, and SAHAND SALAMAT, University of California, San Diego
JUSTIN MORRIS, University of California, San Diego, and San Diego State University
RANGANATHAN RAMKUMAR, JEFFREY YU, ANIKET TIWARI, and JAEYOUNG KANG, University of California, San Diego
MOHSEN IMANI, University of California, San Diego, and University of California, Irvine
BARIS AKSANLI, San Diego State University
TAJANA ŠIMUNIĆ ROSING, University of California, San Diego

Processing large amounts of data, especially in learning algorithms, poses a challenge for current embedded computing systems. **Hyperdimensional (HD) computing (HDC)** is a brain-inspired computing paradigm that works with high-dimensional vectors called *hypervectors*. HDC replaces several complex learning computations with bitwise and simpler arithmetic operations at the expense of an increased amount of data due to mapping the data into high-dimensional space. These hypervectors, more often than not, cannot be stored in memory, resulting in long data transfers from storage. In this article, we propose Store-n-Learn, an in-storage computing solution that performs HDC classification and clustering by implementing encoding, training, retraining, and inference across the flash hierarchy. To hide the latency of training and enable efficient computation, we introduce the concept of *batching* in HDC. We also present on-chip acceleration for HDC encoding in flash planes. This enables us to exploit the high parallelism provided by the flash hierarchy and encode multiple data points in parallel in both batched and non-batched fashion. Store-n-Learn also implements a single top-level FPGA accelerator with novel implementations for HDC classification training, retraining, inference, and clustering on the encoded data. Our evaluation over 10 popular datasets shows that Store-n-Learn is on average 222× (543×) faster than CPU and 10.6× (7.3×) faster than the state-of-the-art in-storage computing solution, INSIDER for HDC classification (clustering).

CCS Concepts: • **Information systems** → **Flash memory**; • **Hardware** → **Biology-related information processing**; *Emerging architectures;*

Additional Key Words and Phrases: Hyperdimensional computing, in-storage computing, classification, clustering

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC-Global Research Collaboration grants, and also NSF grants 1527034, 1730158, 1826967, 1911095, and 2003279.
Authors' addresses: S. Gupta, B. Khaleghi, S. Salamat, R. Ramkumar, J. Yu, A. Tiwari, J. Kang, and T. Š. Rosing, University of California, San Diego, 9500 Gilman Dr, La Jolla, California, 92093, USA; emails: {sgupta, bkhalegh, sasalama, rramkuma, jey070, artiwari, j5kang, tajana}@ucsd.edu; J. Morris, University of California, San Diego, 9500 Gilman Dr, La Jolla, California, 92093, USA; email: justinmorris@ucsd.edu; M. Imani, University of California, San Diego, 9500 Gilman Dr, La Jolla, California, 92093, USA; email: m.imani@uci.edu; B. Aksanli, San Diego State University, 5500 Campanile Dr, San Diego, California, 92182, USA; email: baksanli@sdsu.edu.

1539-9087/2022/07-ART22 $15.00
https://doi.org/10.1145/3503541

ACM Reference format:
Saransh Gupta, Behnam Khaleghi, Sahand Salamat, Justin Morris, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Jaeyoung Kang, Mohsen Imani, Baris Aksanli, and Tajana Šimunić Rosing. 2022. Store-n-Learn: Classification and Clustering with Hyperdimensional Computing across Flash Hierarchy. *ACM Trans. Embedd. Comput. Syst.* 21, 3, Article 22 (July 2022), 25 pages.
https://doi.org/10.1145/3503541

1 INTRODUCTION

The Internet of Things, the ever-increasing demand for new complex applications, and slowdown of Moore's law have pushed current processing systems to their limits. Running data-intensive workloads with large datasets on traditional cores results in high energy consumption and slow processing speed. Moreover, most Internet of Things applications today use state-of-the art machine learning algorithms that have severe energy requirements. With the growing importance of achieving energy efficiency, there is a need to explore emerging computation models.

Brain-inspired **Hyperdimensional Computing (HDC)** is a computation paradigm that represents data in terms of extremely large vectors, called *hypervectors*. These hypervectors may have tens of thousands of dimensions and present data in the form of a pattern of signals instead of numbers. By representing data in high-dimensional space, HDC reduces the complexity of operations required to process data. HDC builds upon a well-defined set of operations with random HDC vectors, making HDC extremely robust in the presence of failures, and offers a complete computational paradigm that is easily applied to learning problems [25]. Prior work has shown the suitability of HDC for various applications like activity recognition, face detection, language recognition, and image classification, among others [12–14, 17, 28, 36, 38].

Although HDC provides improvements in performance and energy consumption over conventional machine learning algorithms, it still involves fetching each and every data from memory/disk and processing it on CPUs/GPUs. Today, extremely large datasets are stored on disks. In addition, the humongous amount of data generated while running HDC cannot always be fit into the memory, eventually killing the process. Recent work has introduced computing capabilities to **solid-state disks (SSDs)** to process data in storage [8, 24, 29, 39]. This not only reduces the computation load from the processing cores but also processes raw data where it is stored. However, the state-of-the-art **in-storage computing (ISC)** solutions either utilize a single big accelerator for an SSD or limit the gains by using complex power-hungry accelerators down the storage hierarchy [31]. Such architectures are not able to fully leverage its hierarchical design.

In this article, we propose an HDC system that spans multiple levels of the storage hierarchy. We exploit the internal bandwidth and hierarchical structure of SSDs to perform HDC operations over multiple data samples in parallel. Our main contributions are as follows:

- We present a novel ISC architecture for HDC that performs HDC classification and clustering completely in storage. It enables computing at multiple levels of SSD hierarchy, allowing for highly parallel ISC. Our hierarchical design provides parallelism and hides a significant part of the performance cost of ISC in the storage read/write operations.
- We introduce the concept of batching in HDC and utilize it to make our ISC implementation more efficient. During training, we batch together multiple data samples encoded in the HDC domain in storage. This allows us to partially process data without accessing all encoded hypervectors. Batching enables us to have a minimal aggregation hardware requirement. Batching also reduces the amount of data sent out of storage.
- Store-n-Learn utilizes die-level accelerators to convert raw data into hypervectors locally in all the flash planes in parallel. Unlike previous work [31], our accelerator is simpler and hides

its computation latency by the long read times of raw data from flash arrays. Our die-level accelerators can perform both batched and non-batched encoding efficiently in flash planes. For batched encoding, the accelerator processes multiple inputs in a page in parallel, whereas it generates multiple dimensions corresponding to an input in parallel during non-batched encoding. This flexibility is enabled by our innovative adder tree design.

- We present a top-level SSD accelerator, which aggregates the data from different flash dies. This accelerator is implemented on an FPGA-based device controller. We implement new and efficient FPGA designs for HDC training, retraining, inference, and clustering. Although HDC training provides sufficiently accurate initial models, retraining significantly improves the accuracy of the models by iterating over training data and updating the models multiple times. Store-n-Learn inference allows the users to directly obtain the classification result from the storage drive without sending the entire model to the host. Moreover, Store-n-Learn clustering leverages the FPGA already present in storage and iteratively processes the datasets multiple times to generate high-quality cluster centers.
- We also present host-side and drive-side primitives to enable the FPGA to work seamlessly with the die-level accelerators.
- We evaluate Store-n-Learn over 10 popular classification and clustering datasets. Our experimental results show that Store-n-Learn is on average 222× (543×) faster than CPU and 10.6× (7.3×) faster than the state-of-the-art ISC solution, INSIDER for HDC classification (clustering).

2 RELATED WORK

Hyperdimensional computing. Prior work applied the idea of HDC to a wide range of learning applications, including language recognition [37], speech recognition [15], gesture detection [34], human-brain interaction [35], and sensor fusion prediction [38]. For example, Rahimi et al. [36] proposed an HD encoder based on random indexing for recognizing a text's language by generating and comparing text hypervectors. In another work, Rahimi et al [34] proposed an encoding method to map and classify biosignal sensory data in high-dimensional space. Imani et al. [12] proposed a general encoding module that maps feature vectors into high-dimensional space while keeping most of the original data. Prior work also designed different training framework to enable sparsity and quantization in HDC [11, 21]. Prior work also tried to design different hardware accelerators for HDC. This included accelerating HDC on existing FPGA, ASIC, and processing in-memory platforms [19, 41, 45]. However, these solutions do not scale well with the number of classes and dimensions, primarily due to the data movement issue. In addition, the existing processing in-memory architectures only accelerate the encoding, training, or associative search, and they are not scaled with the number of classes of hypervector dimensions. Moreover, they work with a binary hypervector, which has been shown to provide very low classification accuracy in HD space [16]. In contrast, our proposed Store-n-Learn accelerates all the phases of HDC classification and clustering by fundamentally addressing data movement and memory requirement issues. In addition, Store-n-Learn scales with the size of data and the complexity of the learning task.

In-storage computing. The major bottlenecks in the current storage systems include the slow flash array read latency and the SSD to host I/O latency [30]. To alleviate these issues, prior work introduced ISC architectures [29, 42]. These works exploited the embedded cores present in the SSD controller to implement ISC. Another set of works [8, 24, 31] used ASIC accelerators in SSD for specific workloads. Ruan et al. [39] proposed a full-stack storage system to reduce the host-side I/O stack latency. All of these works propose single-level computing in storage; however, Store-n-Learn is the first work to push the computing all the way down to the flash die to extract maximum

parallelism. It also uses a top-level accelerator to provide an additional layer of computing. The combination provides a faster implementation that overcomes the SSD to host transfer bottleneck for HDC.

3 HYPERDIMENSIONAL COMPUTING

Brain-inspired HDC has been proposed as the alternative computing method that processes the cognitive tasks in a more light-weight way [25, 37]. HDC offers an efficient learning strategy without overly complex computation steps such as back propagation in neural networks. HDC works by representing data in terms of extremely large vectors, called *hypervectors*, on the order of 10,000 dimensions. HDC has been shown to incur minimal error rates, providing accuracy similar to state-of-the-art learning algorithms like DNNs [10] and k-means [14]. However, the high-dimensional space of HDC makes it robust to external noise sources and hardware-induced errors like device failures [26], stuck-at-fault errors [45], errors from low-precision hardware [37], and noisy communication [6]. Hence, in noisy and error-prone systems HDC proves superior to algorithms like DNNs and k-means that incur large accuracy losses. HDC performs the learning task after mapping all training data into the high-dimensional space. The mapping procedure is often referred to as *encoding*. Ideally, the encoded data should preserve the distance of data points in the high-dimensional space. For example, if a data point is completely different from another one, the corresponding hypervectors should be orthogonal in the HDC space. There are multiple encoding methods proposed in literature [12, 36]. These methods have shown excellent classification accuracy for different data types. In the following, we explain the details of HDC classification steps.

3.1 Encoding

Let us consider an encoding function that maps a feature vector $\mathbf{F} = \{f_1, f_2, \ldots, f_n\}$, with n features ($f_i \in \mathbb{N}$) to a hypervector $\mathbf{H} = \{h_1, h_2, \ldots, h_D\}$ with D dimensions ($h_i \in \{0, 1\}$). We first generate a projection matrix **PM** with D rows, and each row is a vector with n dimensions randomly sampled from $\{-1, 1\}$. This matrix is generated once offline and is then used to encode all of the data samples. We generate the resulting hypervector by calculating the matrix vector multiplication product of the projection matrix with the feature vector:

$$\mathbf{H}' = \mathbf{PM} \times \mathbf{F}. \tag{1}$$

After this step, each element h_i of a hypervector $\mathbf{H}'$ has a non-binary value. In HDC, binary (bipolar) hypervectors are often used for the computation efficiency. We thus obtain the final encoded hypervector by binarizing it with a sign function ($\mathbf{H} = sign(\mathbf{H}')$) where the sign function assigns all positive hypervector dimensions to 1 and zero/negative dimensions to −1. The encoded hypervector stores the information of each original data point with D bits.

3.2 Training for Classification

In the training step, we combine all of the encoded hypervectors of each class using element-wise addition. For example, in an activity recognition application, the training procedure adds all hypervectors that have the "walking" and "sitting" tags into two different hypervectors. Where $H_j^i = \langle h_D, \ldots, h_1 \rangle$ is encoded for the j^{th} sample in the i^{th} class, each class hypervector is trained as follows:

$$C^i = \sum_j H_j^i = \langle c_D^i, \ldots, c_1^i \rangle. \tag{2}$$

3.3 Classification Retraining

HD classification training requires only a single pass over training data and delivers reasonable accuracy. However, some critical applications and/or situations may demand higher accuracy. In such cases, HD classification retraining can significantly improve the accuracy of the base trained hypervectors by iterating over the training data multiple times. Considering a training input hypervector H_x that belongs to class j but is incorrectly assigned to class k, the retraining step proceeds as follows:

$$C_i = C_i - H_x, \tag{3}$$

$$C_j = C_j + H_x. \tag{4}$$

This step can be repeated several times for the whole dataset until the desired accuracy is achieved.

3.4 Classification Inference

The main computation of inference is the encoding and associative search. We perform the same encoding procedure to convert a test data point into a hypervector, called a *query hypervector*, $Q \in \{-1, 1\}^D$. Then, HDC computes the similarity of the query hypervector with all k class hypervectors, $\{C_1, C_2, \ldots, C_k\}$. We measure the similarity between a query and an i^{th} class hypervector using $\delta\langle Q,\ C_i\rangle$, where δ denotes the similarity metric. The similarity metric most commonly used is cosine similarity, as it provides the highest accuracy. However, other similarity metrics like dot product and Hamming distance for binary class hypervectors are also used. After computing all similarities, each query is assigned to a class with the highest similarity.

3.5 Clustering

The HD clustering algorithm is very similar to the popular k-means algorithm. HD clustering, like k-means, first starts off with random centers. Each cluster center is assigned a unique hypervector. Then, the algorithm iterates through all of the data points while comparing their corresponding hypervectors with those of the cluster centers using the cosine similarity metric. Each data point is assigned the center with maximum similarity. After all points are labeled, the new centers are chosen by superimposing the corresponding points to form an updated set of cluster centers:

$$C_k^{t+1} = \sum_{H_X \in C_k^t} H_X, \tag{5}$$

where $H_X \in C_k^t$ indicates the set of all data points assigned to the cluster represented by C_k after iteration t. The process is repeated until convergence or the maximum number of iterations is reached. Convergence occurs when no point is assigned to a different cluster compared to the previous iteration.

3.6 Applications beyond Classification and Clustering

In addition to the classification and clustering workloads discussed in this article, recent works have used HDC for various types of applications. For example, the work on HyperRec [9] implements a recommender system using HDC. GenieHD [27] uses HDC to implement DNA pattern matching. It uses HDC's high dimensionality to encode the entire DNA reference database, making it easier to be searched. It then matches incoming chunks of HDC-encoded DNA with the reference database. Asgarinejad et al. [1] and Imani et al. [18] used HDC to detect seizures and perform real-time health analysis, respectively. In addition, Neubert et al. [32] demonstrated the application of HDC to different robotic tasks like viewpoint invariant object recognition, place recognition, and learning of simple reactive behaviors. To achieve this, they mapped high-dimensional vectors and

operations of HDC to the more widely studied vector symbolic architectures. These works show that HDC has a wide applicability and can be used to perform a range of tasks.

3.7 Challenges

HDC is light-weight enough to run at acceptable speed on a CPU [16]. Utilizing a parallel architecture can significantly speed up the execution time of HDC [19]. However, with the constantly increasing data sizes along with the explosion in data that occurs due to HDC encoding, running this algorithm on current systems is highly inefficient. All of these platforms need to fetch the extremely large hypervectors from memory/disk to process them. They also require huge memory space to store HDC hypervectors and train on them. With the available parallelism across thousands of dimensions and simple operations needed, ISC is a promising solution to accelerate HDC encoding and training.

General-purpose ISC solutions partially address the data transfer bottleneck but still are not able to fully exploit the huge internal SSD bandwidth [39]. The state-of-the-art application-specific ISC [31] tries to exploit the internal SSD bandwidth but provides only one level of computing, which fails to accelerate applications that either (i) have a computing logic that is too complex to implement using the small accelerator or (ii) require post-processing computation steps. Store-n-Learn aims to overcome these issues by breaking complex HDC algorithms into simpler, both data-size and computation-wise, parallelizable tasks. Then, Store-n-Learn utilizes two levels of computation within the SSD, one at the chip-level and other at the SSD level, to efficiently implement those tasks.

4 STORE-N-LEARN DESIGN

Store-n-Learn is an ISC design that performs HDC classification and clustering completely in storage. It utilizes a two-level computing architecture. The first level encodes the raw data into HDC data, whereas the second level processes this high-dimensional data. The expansion of data size during encoding requires high bandwidth between the first and the second level of computing. In addition, the second level of computing should be able to implement a variety of different computing kernels based on the target application. To achieve that, we implement the first level of computing at the flash chips, whereas the second level of computing is at the SSD controller in the form of an FPGA. Although computing at flash chips can utilize the high internal parallelism (multiple flash chips in SSD hierarchy) and internal bandwidth (higher channel bandwidth vs the SSD output bandwidth), the FPGA allows us to implement configurable computing kernels.

Figure 1 shows an overview of the Store-n-Learn SSD architecture. A flash die consists of multiple flash planes, each of which generates a page during a read cycle. Store-n-Learn inserts a simple low-power accelerator, the die-level accelerator (in green on the right in Figure 1), in each plane to encode every read page into a hypervector. These hypervectors are then sent to a top-level FPGA, which accumulates these hypervectors in batches (in green on the bottom left in Figure 1). The FPGA is also used for retraining, inference, and clustering on the encoded hypervectors received from the flash planes. Store-n-Learn uses a scratchpad (in green on the top left in Figure 1) in the controller to store the projection matrix, which it receives as an application parameter from the host. Batching ensures that data generated by each SSD-wide read operation is used in training as soon as it is available, without waiting for the remaining data.

4.1 Batched HDC Training in Store-n-Learn

The size of raw data (number of data points) combined with the size of each hypervector (size of each encoded data point) makes it unrealistic to store all of the encoded hypervectors and then perform HDC training over them. Hence, we employ batching to perform partial training with the

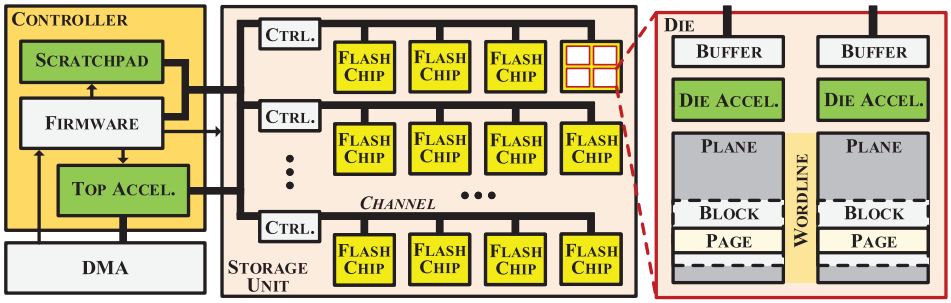

Fig. 1. Store-n-Learn SSD overview. The components added by Store-n-Learn are shown in green.

hypervectors available at any given moment. As mentioned in Section 3, the initial HDC training algorithm to create a class hypervector (2) is to add up all of the encoded samples belonging to a given class. This summation can be split up into batches of partial sums and maintain the same result. For example, say there are s samples for each class—the total sum can be split up into k partial sums or batches and the batch size defined as $b = s/k$, as shown in Equation (6).

$$C^i = \sum_{j=1}^{b} H_j^i + \sum_{j=b+1}^{2b} + \cdots + \sum_{j=((s-1)b)+1}^{s} H_j^i \tag{6}$$

Batching allows Store-n-Learn to process a subset of encoded hypervectors together. Store-n-Learn chip-level accelerators encode raw data into hypervectors and send them to the top-level SSD FPGA accelerator for further processing. All flash chips operate in parallel to encode some of their data, send the hypervectors to the FPGA, and operate on the next set. Each of these hypervectors belongs to a specific *class*. For an application with C classes, we allocate enough memory in the top-level accelerator to store C *model hypervectors*, each assigned to a class. We batch all incoming hypervectors from flash that belong to the same class together and bundle the result with the corresponding model hypervector. This is continued until all required data has been encoded and used to train model hypervectors. In the end, the top-level model hypervectors represent a fully trained model of the data. Batching provides us with two benefits. First, it minimizes the memory requirement during training. Second, it reduces its effective latency by combining hypervectors as soon as they are generated. This hides a major part of training latency with the time taken to read data from flash.

What if the size of model hypervectors is too large to store at top-level FPGA accelerator? Some applications may need too many dimensions or have too many classes to store all model hypervectors at the FPGA, which at best may have few megabytes of **blocked RAMs (BRAMs)**. In such a case, even with balanced data, it will not be possible to train the model completely in storage. However, Store-n-Learn can still perform training in batches and reduce the amount of data sent to the host for processing. Now, instead of allocating FPGA BRAMs for all model hypervectors, it is dynamically allocated according to the encoded input hypervectors available at a time. If an input hypervector does not belong to one of the present models, a model hypervector is sent out to the CPU host and an empty model hypervector corresponding to the class associated with the incoming hypervector is allocated instead. The implementation details are presented in Section 4.3. The host is then responsible for combining various batched training hypervectors together.

In this operating mode, Store-n-Learn still reduces the amount of data movement compared to sending the raw low-dimensional data. Here, we define n as the number of features or dimensionality of the original data, D, as the dimensionality of the encoded hypervectors, and b as the batch

size. When $nb > D$, the total data movement of the resulting batched hypervectors is less than the amount of original data sent in low-dimensional space when the batched hypervector uses the same bitwidth as the original data. However, we can utilize lower bitwidth representations as we encode the data into a hypervector whose elements are $\{-1, 1\}$ and then bundle the hypervectors with element-wise addition. Therefore, the range of data in any given dimension can be defined by the normal distribution with a mean of 0 and standard deviation of $\sqrt{b}$. We can represent each dimension of the batched hypervector with $(\log_2 4\sqrt{b}) + 1$ bits while maintaining an accurate representation. We multiply by 4 to capture 4 standard deviations away and add 1 to account for the sign bit. In this case, assuming the original data is represented with 32 bits, Store-n-Learn sends less data than the data movement required to send the original data in low-dimensional space when $32nb > D(\log_2(4\sqrt{b}) + 1)$.

4.2 Encoding Near Data via Flash Hierarchy

The modern SSD architecture is hierarchical in nature. An SSD has multiple channels. Each channel is shared by four to eight flash chips as shown in Figure 1. The flash chip may consist of several flash dies that are further divided into flash planes, each plane consisting of a group of blocks, each of which store multiple pages. Each plane has a page buffer to write the data to. Operations in the SSD happen in page granularity where the size of pages usually ranges from 2 to 16 KB [4]. Throughout this article, "B" represents bytes. To fully utilize the flash hierarchy, we introduce accelerators for each flash plane as shown in Figure 1. The aim of this added computing primitive is to process the data where it has no conflict or competition for resources.

4.2.1 Chip-Level Accelerator Design. The Store-n-Learn plane accelerator encodes an entire page with raw data to generate a D-dimensional hypervector. Let us assume the SSD page size to be 4 KB (p_s) with each data point being 4 bytes (d_s). This translates to 1K data points (p_s/d_s). Let the feature vector contain 1K features. Assuming that the feature vectors are page aligned, each page stores one feature vector. HDC encoding multiplies an n-size feature vector with a projection matrix containing $D \times n$ 1-bit elements. Our accelerator calculates the dot product between two page-long vectors, one read from the flash array and another being a row vector of the projection matrix. This involves element-wise multiplication of the two vectors and adding together all elements in the product. Since the weights in the projection matrix $\in \{1, -1\}$, we reduce the bits required to store the weights by mapping them such that $1 \rightarrow 1$ and $(-1) \rightarrow 0$. We use 2's complement to break the multiplication into an inversion using XNOR gates and then adding the total number of inverted inputs to the accumulated sum of XNOR outputs. The accelerator is shown in Figure 2. It consists of an array of 32K XNOR gates followed by a 1K input tree adder (labeled CSA in Figure 2). The tree adder is a pruned version of the Wallace carry save tree, where the operand size throughout the tree is fixed to 4B. It reduces 1,024 inputs to 2, which is followed by a carry look-ahead addition (labeled CLA in Figure 2). This gives us the dot product of the two vectors. It is the value of one dimension of the encoded hypervector. The accelerator is iteratively run D times to generate D dimensions. Depending upon the power budget, Store-n-Learn may employ multiple parallel instances of this accelerator to reduce the total number of iterations. Since D is generally large, the generated D-dimensional vector is multi-page output. Store-n-Learn writes the output of the accelerator to the page buffer of the plane, which serves as the response to the original SSD read request.

4.2.2 Storing Input Data. The preceding accelerator assumed the size of the feature vectors to be exactly the same as that of a page. However, this is rarely the case. State-of-the-art ISC designs use page-aligned feature vectors, which may lead to poor storage utilization if the feature vector

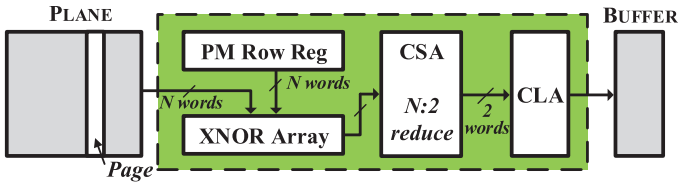

Fig. 2. Store-n-Learn die accelerator.

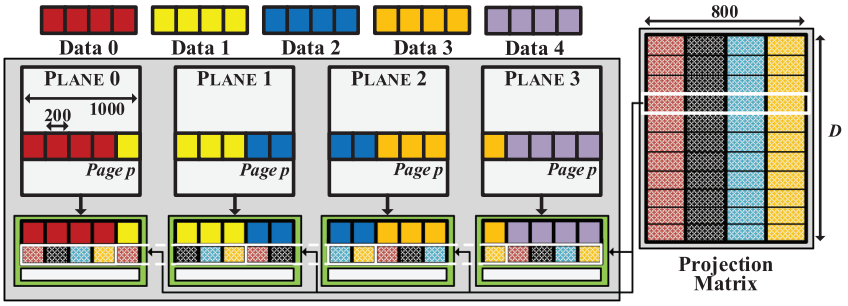

Fig. 3. Data storage scheme in Store-n-Learn and the corresponding segmentation of the projection matrix. Data represents a feature vector.

size is too small or just larger than the page size. For example, in a page-aligned feature vector setting, a 4-KB page may fit only one 512-B feature instead of eight. In addition, a 5-KB feature vector may occupy two complete pages. To alleviate the issue, we propose a cross-plane storing scheme, which considers all planes in a chip when storing data, with the goal of increasing the traditional ISC storage utilization while being accelerator-friendly. We first describe the case when the size of the feature vector is smaller than the page size. The scheme, shown in Figure 3 on the left, divides an n-sized feature vector into n_p equal segments such that the most efficient storage is given when

$$\underset{c}{\text{argmax}} \left(c \times n \; + \; d.n/n_p \leq p_s\right),$$

where c is the number of complete n-sized feature vectors in a p_s-sized page, n_p is the number of planes per chip, and $d \in \{0, 1, \ldots n_p\}$. Hence, a page would contain $c \times n_p + d$ segments in total. Having n_p equal segments instead of any variable segmentation allows the accelerator to have a simple segment-wise weight allocation. Each row vector in the projection matrix of a plane is divided into the same-sized segments as the feature vector as shown in Figure 3 on the right. This allows Store-n-Learn to increase storage efficiency while minimizing the control overhead of the accelerator.

If the size of the feature vector is less than the page size, Store-n-Learn uses the same segmentation size. However, the number of segments in a page are given by

$$\underset{d}{\text{argmax}} \left(d.n/n_p \leq p_s\right).$$

A drawback of this scheme is that individual reads for small feature vectors may require reading two pages instead of one. However, our main purpose is to obtain trained vectors and not raw feature vector values. Moreover, since a feature split across two planes shares the same block and page number, they are both read at the same time.

4.2.3 Encoding on Store-n-Learn Flash Chips. Although the new data storing scheme improves the page utilization, it does not suit well the chip-level accelerator. As proposed before, our accelerator is a dot-product engine. It processes an entire page from the flash array to generate values of different dimensions of the corresponding hypervector. In the new data storage scheme, this would result in an encoded hypervector consisting of multiple and also partial feature vectors. An easy fix would be to just process one feature vector at a time by setting the remaining inputs of the accelerator to 0. However, this would increase the total latency of the accelerator. The situation is worse if the size of feature vectors is very small. We address this problem by extending the concept of batching in Store-n-Learn.

As detailed in Section 4.1, a set of encoded hypervectors can be added dimension-wise without interfering with the HDC training process as long as they belong to the same final trained hypervector, such as the same class model. An encoded dimension (d_i) of a feature vector (FV) is obtained by a dot product between the feature values (FV_i) and the corresponding row of the projection matrix (PM)—that is,

$$d_i = FV_0 \times PM_{i,0} + FV_1 \times PM_{i,1} + \cdots FV_{n-1} \times PM_{i,(n-1)}.$$

Now, to add multiple feature vectors together, we just need to make sure that an element in a feature vector is being multiplied with the corresponding weight of the projection matrix. In that case, we would achieve the same effect as batching, only at a lower level of abstraction. This also works when we have partial features. In this case, the encoded hypervector for the current page would just have partial information and may not correctly represent the data. Some part of this information is contained in the encoded hypervector of another page. However, all of these hypervectors will be added together during training. Hence, the final hypervector will contain all of the information.

To support this strategy in the Store-n-Learn accelerator, the flash controller segments the projection matrix in the same way as the feature vectors in the planes and sends the corresponding segments to the accelerator in each plane. It is important to note that only the features belonging to the same class are added together in batches. Thus, a chip-level accelerator performs a bitwise comparison between the labels of feature vectors in a page and only processes those belonging to the same final model together.

4.2.4 Encoding without Batching. The encoding acceleration discussed previously works well while training class hypervectors for classification because training input samples can be batched together. However, other tasks like clustering, retraining, and inference operate on individual data samples. Hence, they cannot utilize batched hypervectors and require access to individual ones.

As discussed before, encoding individual data points is slow and does not fully utilize the adder tree present in the encoding accelerator. Hence, unlike batched encoding where we could get away with generating just one dimension per iteration of the accelerator, here we need to generate multiple dimensions in parallel. Since each dimension is independent, one way to improve the latency of encoding individual vectors would be to introduce multiple adder trees, each computing one dimension. However, this would linearly increase the power and area of the accelerator. Moreover, the optimal size and number of trees would differ for each application.

Instead, we preserve the current single adder tree and introduce **carry look-ahead adders (CLAs)** at intermediate stages as shown in Figure 4. The figure shows only a part of our complete 20-stage adder tree. Our 32-bit CLA implementation has a latency similar to four sequential carry save additions—that is, four stages of the carry save adder (CSA) tree. Hence, we generate our tree using 4-stage CSAs. We also add CLAs after every 4 stages, as shown with blue and purple boxes in Figure 4. For example, a 16 (20)-stage CSA consists of 113 (455) smaller and independent 4-stage,

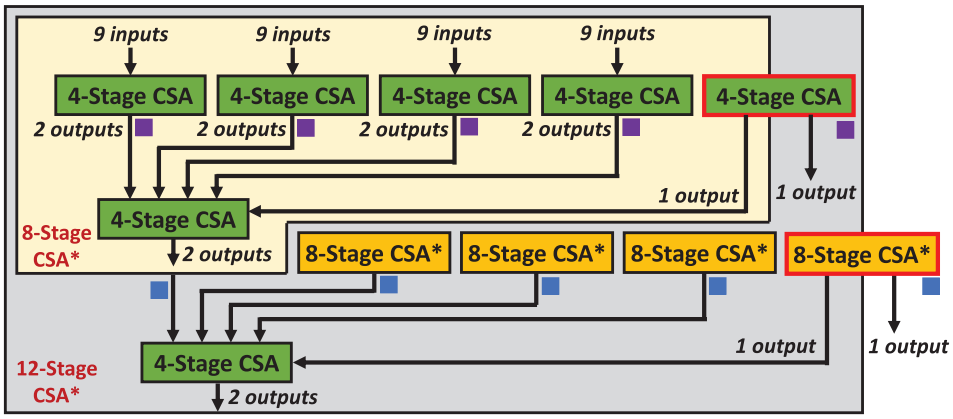

Fig. 4. Modified CSA in the die accelerator to encode individual feature vectors in Store-n-Learn. The blue and purple squares represent intermediate CLAs.

24 (97) 8-stage, 4 (19) 12-stage, and 1 (3) 16-stage CSAs. We insert a 32-bit CLA for each of these independent CSAs. This results in a total of 141 (574) intermediate 32-bit CLAs. Each of these CLA-enabled independent trees can generate one output (dimension) each. Hence, in the case when the size of feature vector is significantly smaller than the page size and less than the input size of any of the CLA-enabled smaller trees, these trees can generate a dimension each. Thus, if a feature vector has a size of, say, 32 (8), we utilize the stage 8 (stage 4) CLAs (i.e., the blue (purple) boxes in Figure 4). For a 1,024-input adder, we can generate 24 dimensions of the hypervector corresponding to this feature vector in parallel. To enable this, the feature vector is input to each smaller CSA. Moreover, the projection submatrix corresponding to the 24 dimensions is flattened and supplied to the accelerator. The preceding modification allows us to generate multiple dimensions in parallel, significantly boosting the performance of single-feature vector encoding.

4.3 Accelerating HD Data Processing at the Controller

The encoded hypervectors from flash chips are used for further processing (learning in our case) in the top-level accelerator, which is implemented on an FPGA present in the SSD controller. We use the FPGA because it is flexible with the application parameters and can be configured using the primitives provided by INSIDER [39]. Previous works perform data encoding on the FPGA to avoid storing the encoded data, which requires more storage space. Supporting HD encoding on the FPGA consumes a lot of FPGA resources and thus limits the performance of the accelerator. Store-n-Learn uses flash chips to encode the data in real time that saturates the internal SSD bandwidth. Thus, Store-n-Learn dedicates all FPGA resources for HD training, inference, and retraining, thereby providing higher performance as conventional FPGA-based accelerators. The initial training iteration builds the HD model, a class hypervector for each class. However, to fine-tune the HD model and increase accuracy, multiple retraining iterations may be needed.

4.3.1 Initial Training. In the FPGA, we first allocate memory for the final class hypervectors. For each class, the FPGA has an input queue, where the input hypervectors belonging to that class are indexed, and an accumulator, which serially accumulates the vectors in the input queue to generate the final class hypervector. The introduction of class-wise input queues removes the input data dependency of the accumulator by pre-processing class labels. An accumulator simply needs to read the input index from its queue and operate on the corresponding data. It makes the

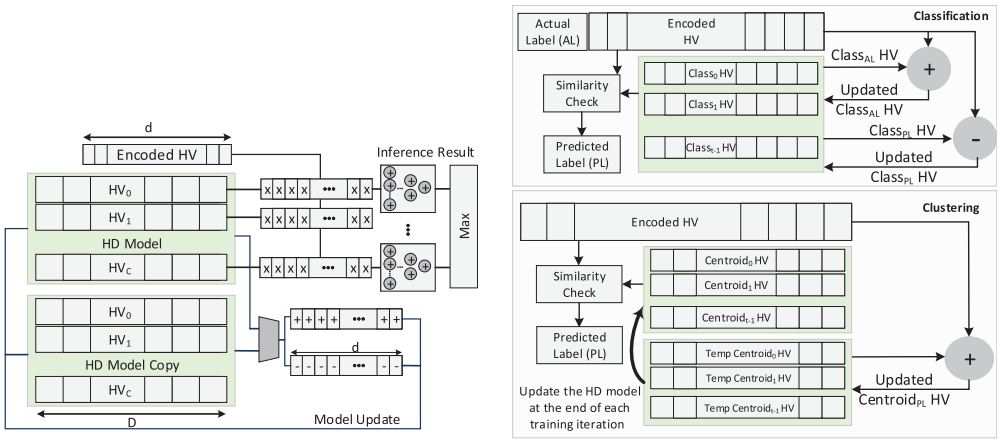

Fig. 5. Store-n-Learn top-level FPGA design. (a) Retraining and inference for HD classification. (b) HD clustering as compared to the retraining step in HD classification.

computation for different classes independent and parallelizable. The accumulators for each class then operate in parallel to add an input hypervector from the queue to the corresponding class hypervector. We divide the hypervectors into partitions to allow partial parallelism.

4.3.2 Retraining and Inference. As explained in Section 3.3, the retraining step consists of reading the encoded hypervectors from the storage, performing the inference, comparing the classification output with the data label, and adjusting the HD model in case of misprediction. To adjust the model, the encoded hypervector is added to the class it belongs to and is subtracted from the mispredicted class hypervector. Figure 5(a) shows the architecture of the Store-n-Learn FPGA-based accelerator for HD retraining and inference. The encoded hypervector is read from the storage device and stored into the *encoded hypervector* buffer.

During HD inference, Store-n-Learn in every clock cycle reads d dimensions of the encoded hypervector, and since HD operations can be parallelized in the dimension level, it calculates the partial similarity metric between the dimensions of the encoded hypervector and corresponding dimensions of the class hypervectors. In HD inference, in every clock cycle, Store-n-Learn calculates the similarity metric between d dimensions of the encoded hypervector and d dimensions of C class hypervectors. For each class, Store-n-Learn performs d multiplications and accumulates the multiplication results in a tree adder with d inputs. At the end, the class with the maximum similarity is the inference result. In each cycle, Store-n-Learn calculates a part of similarity metric and the entire inference is executed in $\frac{D}{d}$ cycles.

To perform HD retraining, Store-n-Learn first performs HD inference and compares the prediction label with the original data label. As illustrated in Figure 5(a), for each misprediction, one addition and one subtraction is needed. Since during the retraining stage an entire encoded hypervector is needed, Store-n-Learn locally stores it on FPGA BRAMs. If Store-n-Learn predicts the label correctly, it reads the next encoded input; otherwise, it performs the model adjustment in $\frac{D}{d}$ cycles.

4.3.3 Clustering. Multiple clustering iterations are required for HD clustering model to converge. In each clustering iteration, HD uses the existing centroids to clusters the input data, and uses the clustered data, at the end of iteration, to update the cluster centroids. Store-n-Learn uses the average of the encoded hypervectors assigned to a cluster as the cluster centroid, giving us

the initial clustering model. This is followed by multiple clustering iterations. For each iteration, Store-n-Learn uses a copy of the latest HD clustering model to update the centroids. In an iteration, Store-n-Learn uses the clustering model to perform a similarity check for each encoded input and assigns a cluster to it. Then, the corresponding input hypervector is added to the predicted cluster centroid of the iteration's copy of the HD clustering model. At the end of each clustering iteration (i.e., after processing all the input data), the HD clustering model is replaced by the updated copy of the HD clustering model.

Figure 5(b) highlights the differences between Store-n-Learn HD retraining and HD clustering. To support HD clustering, Store-n-Learn reuses the similarity check module of HD classification to find the predicted cluster. It also needs a copy of the HD model to update the centroids. As illustrated in the figure, Store-n-Learn requires a duplicate of the HD model memory to support HD clustering, and it reuses the adder array to update the cluster centroids. Therefore, Store-n-Learn supports HD clustering with double BRAM utilization and with minimal logic overhead, only for generating related control signals. In each cycle, similar to classification inference, the similarities between the encoded hypervector and the clustering centroid hypervectors are calculated. Finding the closest cluster takes $\frac{D}{d}$ cycles. Then, Store-n-Learn uses the predicted clusters to update the centroids. Updating the centroids takes another $\frac{D}{d}$ cycles.

4.4 Software Support

The top-level FPGA uses an INSIDER acceleration cluster [39] to implement all HDC operations other than encoding. We utilize INSIDER's software stack to connect Store-n-Learn to the rest of the system. We modify the SSD drivers and the INSIDER virtual files mechanism to enable computing in flash chips and make it visible to the FPGA. Store-n-Learn derives its base system architecture from INSIDER [39]. The INSIDER framework is an API what, while being compatible with POSIX, allows us to implement an ISC accelerator cluster. The INSIDER API takes a C++ or RTL code as an input and programs the acceleration cluster (running on drive FPGA) accordingly. The drive program interface has three FIFOs. The data input (output) FIFO takes in the input (output) data that is needed (generated) by the accelerator. The parameter FIFO contains the runtime parameters for the FPGA that are sent by the host. INSIDER keeps control and data planes of ISC separated. The drive control and standard operations are handled by the SSD firmware while all compute data from flash chips are intercepted by the top-level FPGA accelerator for computing. The FPGA does not care about the source and/or destination of the data.

4.4.1 Store-n-Learn Host-Side Support. The INSIDER API uses POSIX-like I/O functionality to communicate with the driver. INSIDER has a standard block device driver with changes made to the virtual read and write functionalities to accommodate for the programmable accelerator clusters in the drive. However, the current abstraction allow us to pass directive/parameters only to the ISC FPGA and not the drive. We define a new API, `send_mode`, which defines the mode for read and write operations, further discussed in Section 4.4.2. It passes a single integer, *mode*, to the drive firmware while opening a virtual file. For a non-ISC read/write from the drive, *mode* is set to 0. During an ISC read, *mode* represents the *expansion factor* (*EF*). *EF* defines the increase in the size of raw data after encoding. For example, $EF = 5$ means that each page of raw data generates five pages of encoded data (due to large D). In this case, *mode* is set to 5. This parameter is necessary to enable the drive to read the required number of pages from the flash chips. Since *EF* is dependent on the number of features of the data and dimensionality requirement of the application, it remains constant for an entire run. Similarly, a non-zero *mode* signifies ISC write. In this case, the data being sent to the drive contains the elements of the HDC projection matrix

and is written to the controller scratchpad. No data is written to the flash chips. During write, we only care about whether *mode* is zero or non-zero.

4.4.2 Store-n-Learn Drive-Side Architecture. Store-n-Learn implements its top-level accelerator described in Section 4.3 as an INSIDER acceleration cluster, which enables the final training step. However, the INSIDER system does not support Store-n-Learn's die-level acceleration because the standard read/write drive operations cannot readily accommodate on-the-fly change in data size while reading encoded pages and writing projection matrix elements to the on-die accelerator.

Store-n-Learn introduces the processing capability between flash planes and page buffers, but sometimes only raw data may be required. Hence, Store-n-Learn employs two read modes: normal and compute. It uses the die-level accelerator in multiplexed mode where a read page is sent to the accelerator for processing only in compute mode, shown in Figure 6. In normal mode, the plane directly writes the original page to the page buffer. Moreover, response type in the two modes also differs. A normal read results in just one page, whereas a compute read responds with multiple but fixed number of pages. Store-n-Learn uses application specifications such as feature vector size and dimensionality requirement to generate an expansion factor, which is supplied to the SSD firmware by the host, as explained in Section 4.4.1. The firmware uses this factor to calculate the response size for page read commands in compute mode.

Store-n-Learn also employs two write modes: normal and compute. The compute mode is used to supply projection matrix data to the on-die accelerators. In normal mode, data is written in the data buffer and then programmed in the flash array. In compute mode, the data in data buffer is sent to the accelerators as shown in Figure 6. The writes in compute mode are fast since the data is just latched in CMOS registers instead of flash arrays. Unlike a compute mode read, where the same command can be issued to all of the chips, compute mode write requires individual commands for each plane to configure their respective on-die accelerators. This follows from Figure 3. Each plane gets the same segments, but their positions may differ for different planes. A write configuration command is separately issued for each plane. For each plane, it configures the size of segment (seg_S), number of input segments (seg_{in}), actual number of segments in the plane (seg_{act}), and the ID of the first segment (seg_{one}). The format of the command is $[seg_S, seg_{in}, seg_{act}, seg_{one}]$. For example, the command for plane 0 and plane 2 in Figure 3 would be $[200, 4, 5, 0]$ and $[200, 4, 5, 2]$, respectively. Although sequential, this step has negligible latency overhead because it can be performed in parallel for all of the flash chips.

As discussed briefly in Section 4.2, the flash controller sends the projection matrix elements to the respective accelerators. SSD receives the projection matrix from the host. We introduce a dedicated scratchpad in the flash controller to store the matrix. The controller sends the elements in page-sized frames to the die accelerators. The frames consist of multiple segments and are used by the die accelerators according to the configuration command, as shown in Figure 3.

4.5 Store-n-Learn Is More Than Just a HDC Accelerator

We envision Store-n-Learn as a system that, in addition to performing all functions of a standard SSD, can accelerate HDC. However, since Store-n-Learn is based on the INSIDER system stack, it can perform a variety of different computations. The FPGA present in the controller can be configured to perform any computation task on the data read from the drive. The hierarchical nature of Store-n-Learn makes the FPGA and die-level accelerators independent of each other. If the target application/computation does not require die-level accelerators, then the Store-n-Learn can operate in normal read mode and supply raw data to the SSD controller, which the FPGA can compute upon. Moreover, the die-level accelerator that performs HD encoding is essentially a

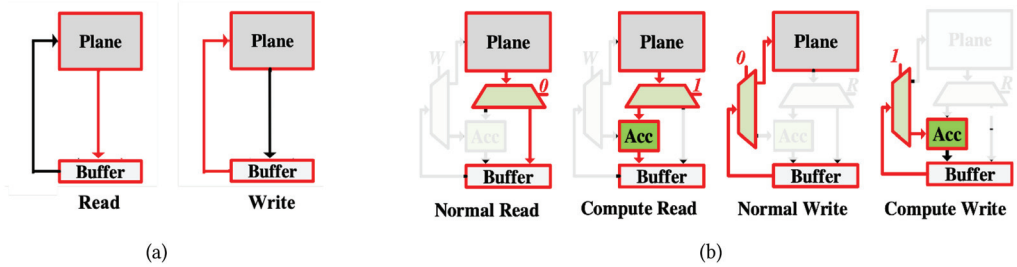

Fig. 6. Different read and write modes in standard SSD (a) and Store-n-Learn (b). The components in red are active during the operation.

dot-product engine. Dot product is one of the major computations in applications like neural networks. Hence, applications other than HDC can use Store-n-Learn either fully or partially.

5 RESULTS

5.1 Experimental Setup

We developed a simulator for Store-n-Learn that supports parallel read and write accesses to the flash chips. We utilized Verilog and Synopsys *Design Compiler* to implement and synthesize our die-level accelerator at 45 nm and scale it down to 22 nm. The top-level FPGA accelerator has been synthesized and simulated in Xilinx Vivado. For Store-n-Learn drive simulation, we assume the characteristics similar to the 1-TB Intel DC P4500 PCIe-3.1 SSD connected to an Intel Xeon CPU E5-2640 v3 host. The parameters for Store-n-Learn are shown in Table 2.

We compare Store-n-Learn with a seventh-generation, 2.4-GHz Kaby Lake Intel Core i5 CPU with 8 MB of RAM and 256 GB of SSD. We also compare it with a 3.5-GHz Intel Xeon CPU E5-2640 v3 CPU server with 256 GB of RAM and 2 TB of local disk. In addition, we compare with a CUDA implementation of the HDC pipeline on an Nvidia GTX 1080 Ti GPU. We also compare Store-n-Learn with INSIDER [39] and DeepStore [31], the state-of-the art ISC solutions. INSIDER is a full-stack storage system and uses a top-level FPGA accelerator in the drive for ISC. DeepStore is an ISC implementation for query-based workloads that employs specialized accelerators in the SSD. For all of our experiments, including those for other ISC solutions, the data is assumed to be channel striped and stored using Store-n-Learn's proposed scheme.

5.2 Workloads

5.2.1 Classification. We evaluate the efficiency of Store-n-Learn on five popular classification applications, as summarized in Table 1 and listed next:

Speech recognition (ISOLET): The goal is to recognize voice audio of the 26 letters of the English alphabet [23].
Face recognition (FACE): We exploit the Caltech dataset of 10,000 web faces [7]. Negative training images (i.e., non-face images) are selected from the CIFAR-100 and Pascal VOC 2012 datasets [5].
Activity recognition (UCIHAR): The dataset includes signals collected from motion sensors for eight subjects performing 19 different activities [44]. *Medical diagnosis* (CARDIO): This dataset provides a medical diagnosis based on cardiotocography information about each patient [3].
Gesture recognition (EMG): The dataset contains EMG readings for five different hand gestures [2].

Table 1. Workload Summary

Dataset	Features	Classes	Batch Size	Dataset	Features	Clusters	Batch Size
ISOLET	617	26	6	**Hepta**	3	7	1,365
FACE	608	2	6	**Tetra**	3	4	1,365
UCIHAR	561	12	7	**TwoDiamonds**	2	2	2,048
CARDIO	21	2	195	**WingNut**	2	2	2,048
EMG	4	5	1,024	**Iris**	4	3	1,024
Synthetic (DS)	512	10	8	**Synthetic (DS)**	512	10	8

Table 2. Store-n-Learn Parameters

Capacity	1TB	**Channels**	32
Page Size	16KB	**Chips/Channel**	4
External BW	3.2GBps	**Planes/Chip**	8
BW/Channel	800MBps	**Blocks/Plane**	512
Flash Latency	53μs	**Pages/Block**	128
FPGA	XCKU025	**Scratchpad Size**	4MB
Avg Power/DA	8mW	**DA Latency**	1.02ns

DA: Die accelerator.

5.2.2 Clustering. We evaluate Store-n-Learn on FCPS, the fundamental clustering problem suite [43], which has been widely used in the literature. We also evaluate HD clustering on the pattern recognition dataset [22]. The specific datasets used are summarized in Table 1 and listed next:

FCPS Hepta [43]: The three-dimensional Hepta dataset consists of seven clusters that are clearly separated by distance, one of which has a much higher density.
FCPS Tetra [43]: The Tetra dataset consists of 400 data points in four clusters that have large intra-cluster distances. The clusters are nearly touching each other, resulting in low inter-cluster distances.
FCPS TwoDiamonds [43]: The data consists of two clusters of two-dimensional points. Inside each "diamond," the values for each data point were drawn independently from uniform distributions.
FCPS WingNut [43]: The WingNut dataset consists of two symmetric data subsets of 500 points each. Each of these subsets is an overlay of equally spaced points with a lattice distance of 0.2 and random points with a growing density in one corner.
Pattern recognition (Iris) [22]: The dataset consists of samples from each of three species of Iris with four features are present from each sample. One class is linearly separable from the other two; the latter are not linearly separable from each other.

5.3 Comparison with CPU and CPU Server

We first compare Store-n-Learn with CPU and CPU-based server running state-of-the-art implementations of HDC classification and clustering over the five datasets with $D = 10k$. In addition, we generate a synthetic dataset with 10 classes and each data sample having 512 features. We vary the size DS (number of data points) of the synthetic dataset from 10^3 to 10^7.

5.3.1 HDC Classification with Single-Pass Training. The runtime of single-pass classification for different platforms is shown in Figure 7. We observe that Store-n-Learn is on average $3,405\times$ and $1,612\times$ faster than CPU and CPU server, respectively. Our evaluations show that the

improvements from Store-n-Learn increases linearly with an increase in the dataset size. This happens because more data samples result in more huge hypervectors to generate and process. In conventional systems, this translates to a huge amount of data transfers between the core and memory. It should be noted that the CPU system runs out of memory while encoding for 10^6 samples and kills the process. The CPU server faces a similar situation for 10^7 samples. In contrast, since Store-n-Learn generates hypervectors (encoding) while reading data out of the slow flash arrays and processes (training) them on the disk itself, there is minimal data movement involved.

Store-n-Learn (No Batch) in Figure 7 shows the latency of HDC training on Store-n-Learn without applying batching. We see that batching reduces the latency of Store-n-Learn on average by 6.5×. The effective speedup due to batching in CARDIO and EMG datasets is significantly less than the batch size. This is because CARDIO and EMG are small datasets and the number of training samples are not enough to bring forth the complete advantage from batching. This is evident from the synthetic data (*DS*), where *DS* = 10^3 samples achieves 4.1× speedup from batching, whereas *DS* = 10^6 samples is able to achieve 7.99× speedup from batching.

Figure 7 also shows the size of raw input data in each case normalized to the size of the corresponding trained class hypervectors. Store-n-Learn only sends class hypervectors from drive to the host, whereas CPU-based systems fetch all data samples from the disk. We observe that the ratio increases linearly with an increase in the data size. In fact, the size of class hypervectors does not change with an increase in data size as long as the number of classes and required dimensions remain the same.

5.3.2 HDC Classification with Retraining. Figure 8 shows the runtime of HD classification with 50 epochs of retraining for different platforms. We observe that Store-n-Learn is on average 222×, 81×, and 28.3× faster than CPU, CPU server, and GPU, respectively. The improvements are lower than those in case of single-pass classification because now FPGA-based retraining, specifically the search component of retraining, is the major latency bottleneck. Our evaluations also show that the performance of Store-n-Learn classification with retraining increases with an increase in either the dataset size or the number of classes. In addition to processing more hypervectors for a larger dataset, more classes increase the total number of the latency critical search operations. Store-n-Learn is able to process much larger datasets than CPU and CPU server, both of which run out of memory while working with 10^6 data samples. The trend for total SSD to host data transfers remains similar to that of single-pass training, where the amount of data transfers saved increases linearly with an increase in the data size.

5.3.3 HDC Clustering. Figure 9 shows the runtime of HDC with 50 epochs of clustering for different platforms. We observe that Store-n-Learn is on average 543× and 187× faster than CPU and CPU server, respectively. Moreover, the latency of Store-n-Learn clustering increases with both dataset size and the number of classes. However, the relative improvements from Store-n-Learn also increase with an increase in dataset size. Store-n-Learn is able to process much larger datasets than CPU and CPU server, both of which run out of memory while clustering 10^6 data samples.

The amount of data transfers saved increases linearly with an increase in the data size. For very small clustering datasets [22, 43], transferring hypervectors of cluster centers instead of raw data increases the data transfers between SSD and host. However, data transfers become a system bottleneck for large datasets, in which case Store-n-Learn significantly reduces the total transfers. For example, Store-n-Learn transfers ~5000× less data compared to CPU-based systems for the synthetic dataset with 1 million samples.

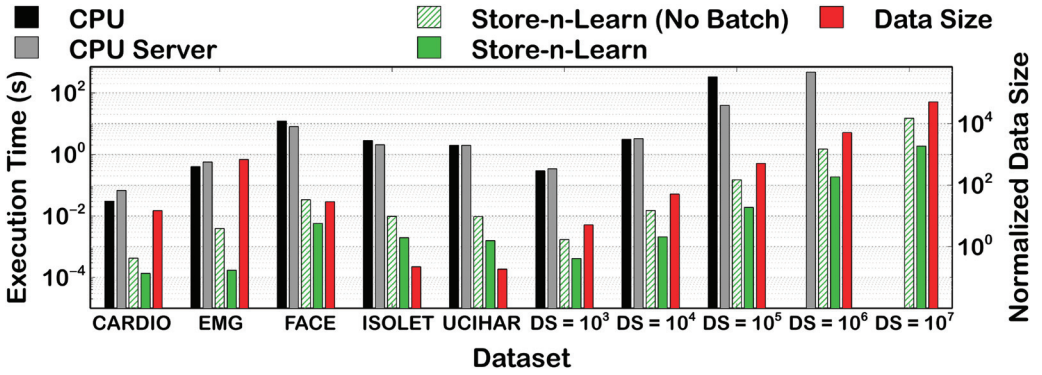

Fig. 7. Runtime comparison of HDC encoding and single-pass classification training in Store-n-Learn with other platforms. The bars in red show the size of raw data normalized to the total size of corresponding class hypervectors in Store-n-Learn. Store-n-Learn (No Batch) represents the latency of Store-n-Learn without applying batching.

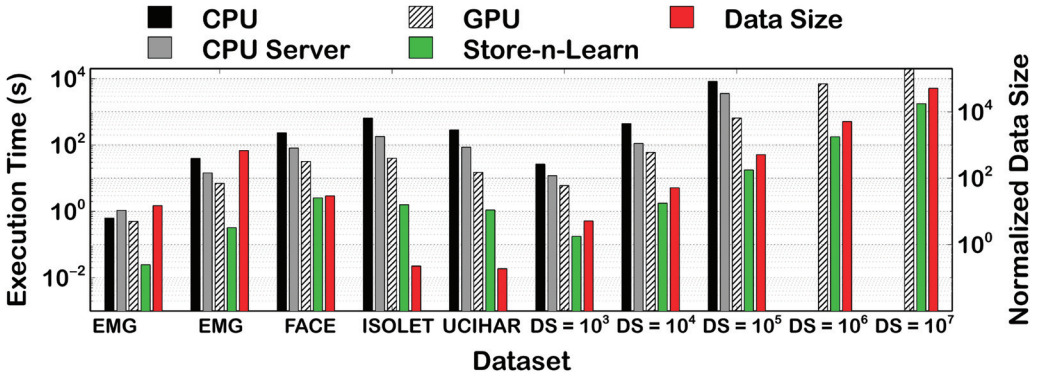

Fig. 8. Runtime comparison of HDC classification with retraining in Store-n-Learn with other platforms. The bars in red show the size of raw data normalized to the total size of corresponding class hypervectors in Store-n-Learn.

5.4 Store-n-Learn Efficiency

Figure 10 shows the breakdown of Store-n-Learn single-pass classification latency normalized to the total latency. Here, I/O shows the time spent in sending the generated class hypervectors to the host. For small datasets, CARDIO and EMG, the latency is dominated by the encoding. However, as the data size increases, the internal SSD channel bandwidth becomes a bottleneck. This indicates that Store-n-Learn is able to completely utilize and saturate the huge internal SSD bandwidth. In addition, a significant amount of time spent in training and some part of the encoding is hidden by the SSD channel latency. As a result, the combined latency is less than sum of the latency for individual stages. For the example of the FACE dataset, even though the training takes more than half of the total latency, a negligible portion of it actually contributes to the overall latency. It shows that Store-n-Learn stages are able to hide some of their latency. This is in contrast to INSIDER [39], where the runtime is dominated by the latency of encoding and training in the FPGA. Figure 11 shows the breakdown of latency of different stages in INSIDER. FPGA-based processing (i.e., encoding and training) takes on average 97% of the total INSIDER runtime. However, in the

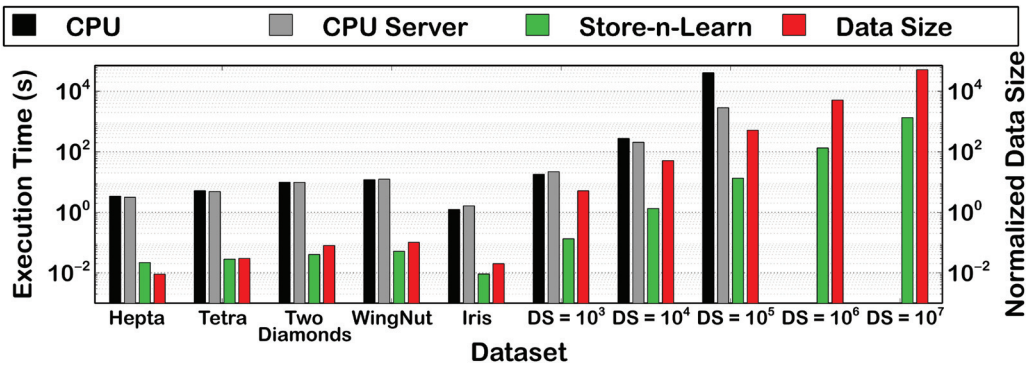

Fig. 9. Runtime comparison of HDC clustering in Store-n-Learn with other platforms. The bars in red show the size of raw data normalized to the total size of corresponding cluster center hypervectors in Store-n-Learn.

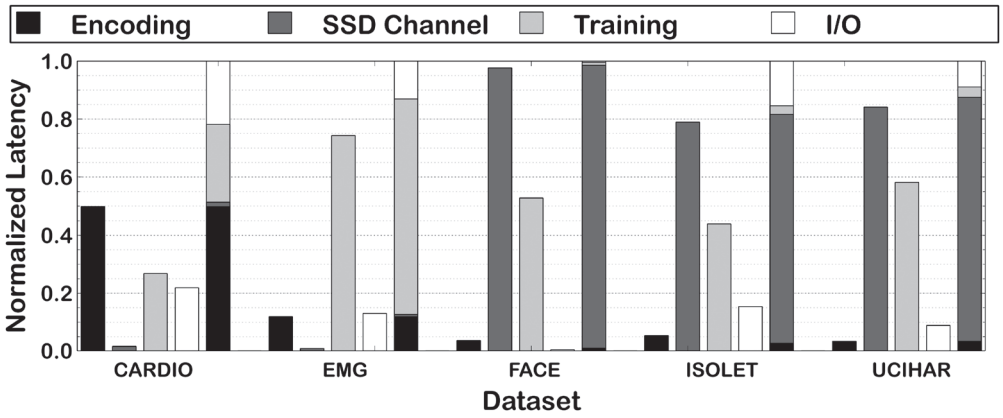

Fig. 10. Breakdown of latency of different stages of HDC single-pass classification normalized to the total latency.

case of HD retraining and clustering, the top-level FPGA accelerator becomes the latency bottleneck. This happens due to the iterative nature of these algorithms.

To demonstrate the scalability provided by Store-n-Learn, we evaluate it over a synthetic dataset with 10^4 samples each with 512 features. We vary the dimensions D from 10^3 to 10^5. Figure 12(a) shows that the latency of Store-n-Learn increases linearly with an increase in the number of dimensions, showing that Store-n-Learn is able to scale with D. Additionally, an increase in D results in longer class hypervectors for the same input data. Hence, the ratio of raw data to hypervector size decreases with an increase in dimensions, falling from from 512 for $D = 1k$ to 2.5 for $D = 10^5$.

We also scale the dataset with the number of classes while keeping its size fixed to 10^4 samples and D as 10^3. Figure 12(b) shows that the Store-n-Learn latency has minor changes with the number of classes when we have fewer than 50 classes. This is because our FPGA has enough resources to train up to 54 classes with $D = 10k$ dimensions. The latency almost doubles for 100 classes. However, when the number of classes increases further, the size of model hypervectors is too large to store in the FPGA. Hence, partially trained hypervectors are then sent to the host for further processing. This can be seen by a jump in the latency for 500 classes in Figure 12(b). In addition to

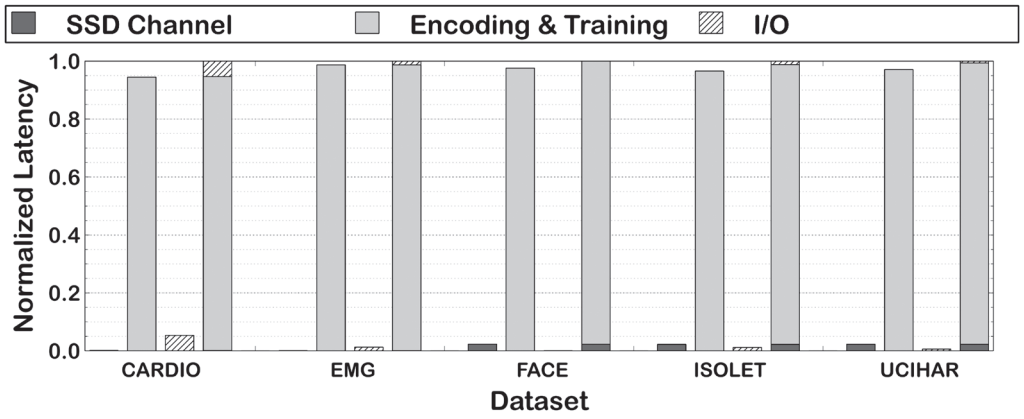

Fig. 11. Breakdown of INSIDER's [39] latency of different stages of HDC single-pass classification normalized to the total latency.

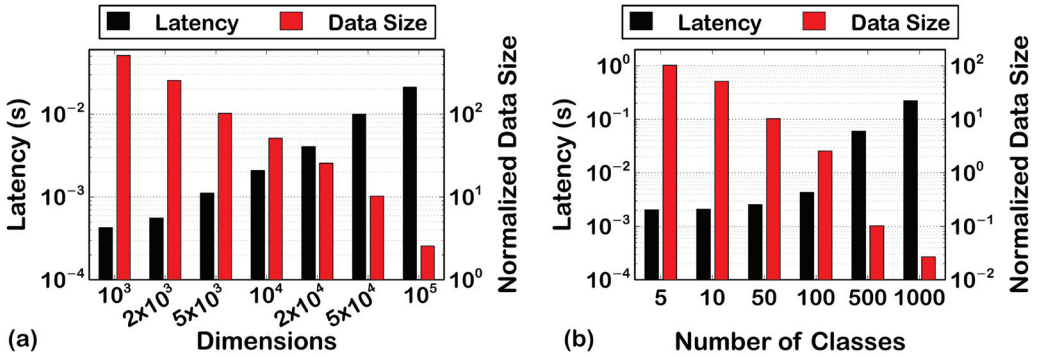

Fig. 12. Change in HDC runtime and raw data size to hypervector ratio with dimensions (a) and number of classes (b).

the time spent in training, transferring the class hypervectors to host creates a major bottleneck. This is also evident from the data size ratio that declines for a large number of classes. A ratio of less than 1 signifies that the size of generated hypervectors is larger than the raw data.

5.5 Store-n-Learn vs Other Algorithms

We compare Store-n-Learn with the best existing algorithms for classification and clustering. For classification, we compare our work with the state-of-the-art DNN network for ISOLET [20]. In our evaluation, Store-n-Learn runs HDC classification with 50 epochs of retraining, whereas DNN is trained on the CPU. We observe that Store-n-Learn is 9.4× faster than DNN while incurring less than 1% accuracy loss. We also compare our design with DNN running on FPGA. No FPGA implementation completely trains DNNs due to the complexity of operations and lack of sufficient on-board resources. Hence, we compare the inference performance of our design with that of DNN running on FPGA [40]. Store-n-Learn is 17.7× faster than FPGA for the ISOLET dataset, with less than 1% accuracy loss.

For clustering, we compare Store-n-Learn with the k-means algorithm [33] for the five clustering datasets on CPU. Store-n-Learn runs HDC clustering with 50 epochs of clustering. The quality

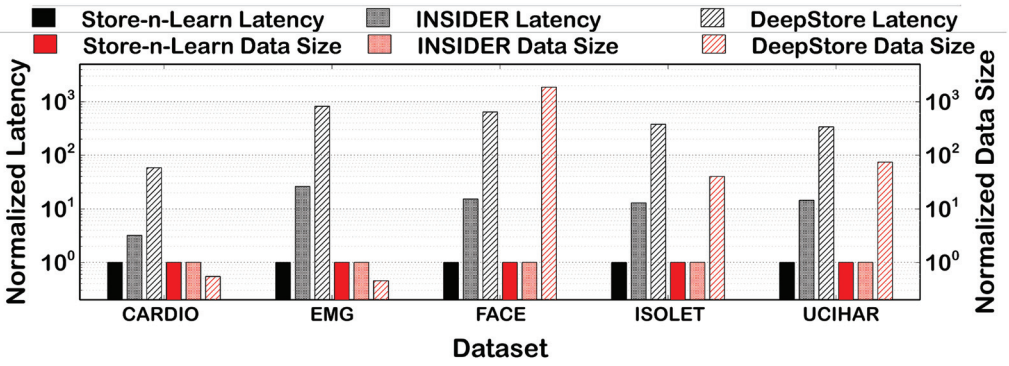

Fig. 13. Runtime and data transfer size comparison of Store-n-Learn classification without retraining with INSIDER [39] and DeepStore [31].

of clustering is measured in terms of mutual information score, which is 1 when the predicted labels are perfectly correlated with the ground truth and 0 when they are totally uncorrelated. Store-n-Learn is on average 1.3× faster than k-means on CPU while providing the same mutual information score. We also compare Store-n-Learn clustering with k-means running on FPGA and observe that Store-n-Learn is 47× faster. Store-n-Learn is faster than the state-of-the-art algorithms for both classification and clustering due to the latency overhead of data transfers in traditional systems. Moreover, the higher complexity of operations in traditional algorithms further makes them slower on FPGA.

5.6 Comparison with Existing ISC Solutions

We compare the performance and data transfer efficiency of Store-n-Learn with state-of-the-art ISC designs INSIDER [39] and DeepStore [31]. In our experiments, INSIDER performs both encoding and training/clustering using the FPGA accelerator in SSD and sends the class hypervectors to the host. Since DeepStore was intended for a completely different application, we replace its accelerator with Store-n-Learn die-level accelerator. During ISC, DeepStore encodes the raw data into hypervectors and sends those hypervectors to the host for training/clustering.

5.6.1 HDC Single-Pass Classification. Figure 13 shows the change in latency and data transfer size of single-pass classification for the three ISC solutions. We observe that Store-n-Learn is on average 14.4× and 446.8× faster than INSIDER and DeepStore, respectively. Although encoding in DeepStore takes the same time as Store-n-Learn, transferring hypervector from SSD to host and further training on them on CPU increases the execution time of DeepStore significantly. However, the SSD channel bottleneck faced by Store-n-Learn is relaxed in the case of INSIDER since it only transfers raw data. However, the FPGA-based HDC encoding+training are on average 21× slower compared to FPGA-based training. In addition, since INSIDER performs training in the SSD, it transfers the same amount of data to the host as Store-n-Learn. However, by transferring untrained hypervectors, DeepStore increases the amount of data transferred on average by 397× compared to Store-n-Learn.

5.6.2 HDC Classification with Retraining. Figure 14 shows the change in latency and data transfer size for complete HDC classification. We observe that Store-n-Learn is on average 10.6× and 179× faster than INSIDER and DeepStore, respectively. For DeepStore, transferring hypervector from SSD to host and further retraining on them for 50 epochs on CPU increases the execution time of DeepStore significantly. INSIDER's FPGA-based HDC encoding and retraining are on

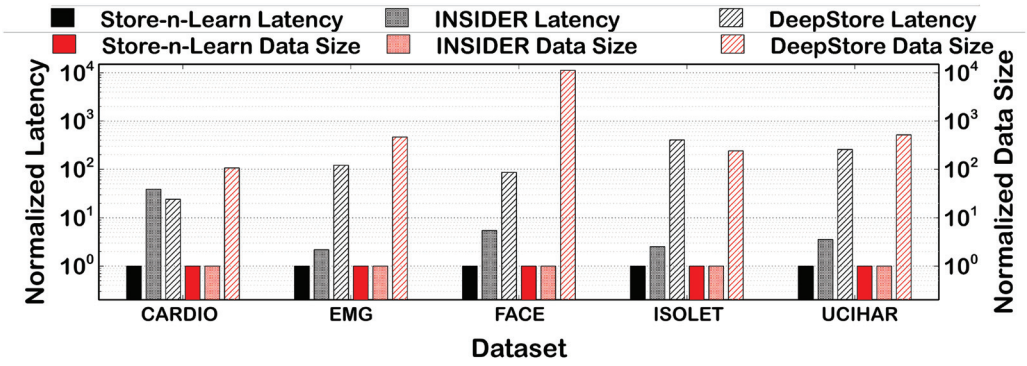

Fig. 14. Runtime and data transfer size comparison of Store-n-Learn classification with retraining with INSIDER [39] and DeepStore [31].

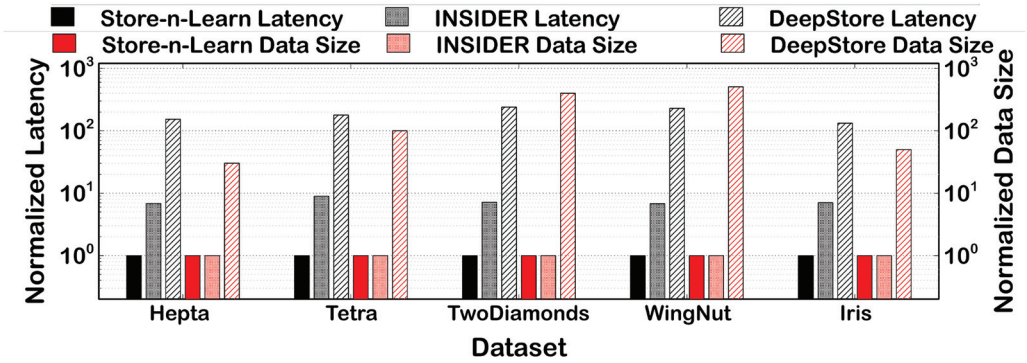

Fig. 15. Runtime and data transfer size comparison of Store-n-Learn clustering with INSIDER [39] and DeepStore [31].

average 10.7× slower compared to Store-n-Learn's FPGA-based retraining because encoding consumes a significant amount of FPGA resources, leaving fewer resources to accelerate latency critical retraining. INSIDER transfers the same amount of data to the host as Store-n-Learn, whereas DeepStore increases the amount of data transferred on average by 2,510× compared to Store-n-Learn. This is 6.3× worse than the data transferred in single-pass classification because in retraining hypervectors are sent for individual data points, eliminating the gains from batched encoding.

5.6.3 HDC Clustering. Figure 15 shows the change in latency and data transfer size for HDC clustering. We observe that Store-n-Learn is on average 7.3× and 187× faster than INSIDER and DeepStore, respectively. The latency results follow the same trends and reasoning as those for HDC classification with retraining. For data transfers, DeepStore increases the amount of data transferred on average by 217× compared to Store-n-Learn. The data transfer overhead of DeepStore worsens with an increase in the dataset size.

6 CONCLUSION

In this article, we proposed an in-storage HDC system that spans multiple levels of the storage hierarchy. We exploited the internal bandwidth and hierarchical structure of SSDs to perform

HDC classification and clustering in-storage. We proposed batched HDC training to enable partial processing of HDC hypervectors. We further proposed a die-level accelerator for HDC encoding and top-level FPGA accelerators for HDC training, retraining, inference, and clustering. Our evaluation shows that Store-n-Learn is on average 222× (543×) faster than CPU and 10.6× (7.3×) faster than the state-of-the-art ISC solution INSIDER for HDC classification (clustering).

REFERENCES

[1] Fatemeh Asgarinejad, Anthony Thomas, and Tajana Rosing. 2020. Detection of epileptic seizures from surface EEG using hyperdimensional computing. In *Proceedings of the 2020 42nd Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'20)*. IEEE, Los Alamitos, CA, 536–540.

[2] Simone Benatti, Elisabetta Farella, Emanuele Gruppioni, and Luca Benini. 2014. Analysis of robust implementation of an EMG pattern recognition based control. In *Proceedings of the 2014 International Conference on Bio-inspired Systems and Signal Processing (BIOSIGNALS'14)*. 45–54.

[3] UCI Machine Learning Repository. 2010. Cardiotocography Data Set. Retrieved February 17, 2022 from https://archive.ics.uci.edu/ml/datasets/cardiotocography.

[4] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, et al. 2018. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *Proceedings of the 2018 IEEE International Solid-State Circuits Conference (ISSCC'18)*. IEEE, Los Alamitos, CA, 338–340.

[5] Mark Everingham, S. M. Ali Eslami, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2015. The Pascal Visual Object Classes Challenge: A retrospective. *International Journal of Computer Vision* 111, 1 (2015), 98–136.

[6] Lulu Ge and Keshab K. Parhi. 2020. Classification using hyperdimensional computing: A review. *IEEE Circuits and Systems Magazine* 20, 2 (2020), 30–47.

[7] Gregory Griffin, Alex Holub, and Pietro Perona. 2007. Caltech-256 Object Category Dataset. Retrieved February 17, 2022 from https://authors.library.caltech.edu/7694/.

[8] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 153–165.

[9] Yunhui Guo, Mohsen Imani, Jaeyoung Kang, Sahand Salamat, Justin Morris, Baris Aksanli, Yeseong Kim, and Tajana Rosing. 2021. HyperRec: Efficient recommender systems with hyperdimensional computing. In *Proceedings of the 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC'21)*. IEEE, Los Alamitos, CA, 384–389.

[10] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. Felix: Fast and energy-efficient logic in memory. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, Los Alamitos, CA, 1–7.

[11] Mohsen Imani, Samuel Bosch, Sohum Datta, Sharadhi Ramakrishna, Sahand Salamat, Jan M. Rabaey, and Tajana Rosing. 2020. QuantHD: A quantization framework for hyperdimensional computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2268–2278.

[12] Mohsen Imani, Chenyu Huang, Deqian Kong, and Tajana Rosing. 2018. Hierarchical hyperdimensional computing for energy efficient classification. In *Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC'18)*. IEEE, Los Alamitos, CA, 1–6.

[13] Mohsen Imani, Yeseong Kim, Sadegh Riazi, John Messerly, Patric Liu, Farinaz Koushanfar, and Tajana Rosing. 2019. A framework for collaborative learning in secure high-dimensional space. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD'19)*. IEEE, Los Alamitos, CA, 435–446.

[14] Mohsen Imani, Yeseong Kim, Thomas Worley, Saransh Gupta, and Tajana Rosing. 2019. HDCluster: An accurate clustering using brain-inspired high-dimensional computing. In *Proceedings of the 2019 Design, Automation, and Test in Europe Conference and Exhibition (DATE'19)*. IEEE, Los Alamitos, CA, 1591–1594.

[15] Mohsen Imani, Deqian Kong, Abbas Rahimi, and Tajana Rosing. 2017. VoiceHD: Hyperdimensional computing for efficient speech recognition. In *Proceedings of the 2017 IEEE International Conference on Rebooting Computing (ICRC'17)*. IEEE, Los Alamitos, CA, 1–8.

[16] Mohsen Imani, John Messerly, Fan Wu, Wang Pi, and Tajana Rosing. 2019. A binary learning framework for hyperdimensional computing. In *Proceedings of the 2019 Design, Automation, and Test in Europe Conference and Exhibition (DATE'19)*. IEEE, Los Alamitos, CA, 126–131.

[17] Mohsen Imani, Tarek Nassar, Abbas Rahimi, and Tajana Rosing. 2018. HDNA: Energy-efficient DNA sequencing using hyperdimensional computing. In *Proceedings of the 2018 IEEE EMBS International Conference on Biomedical and Health Informatics (BHI'18)*. IEEE, Los Alamitos, CA, 271–274.

[18] Mohsen Imani, Tarek Nassar, and Tajana Rosing. 2019. Brain-inspired hyperdimensional computing for real-time health analysis. In *Proceedings of the 2019 IEEE EMBS International Conference on Biomedical and Health Informatics (BHI'19)*. IEEE, Los Alamitos, CA.

[19] Mohsen Imani, Abbas Rahimi, Deqian Kong, Tajana Rosing, and Jan M. Rabaey. 2017. Exploring hyperdimensional associative memory. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, Los Alamitos, CA, 445–456.

[20] Mohsen Imani, Mohammad Samragh Razlighi, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. 2020. Deep learning acceleration with neuron-to-memory transformation. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, Los Alamitos, CA, 1–14.

[21] Mohsen Imani, Sahand Salamat, Behnam Khaleghi, Mohammad Samragh, Farinaz Koushanfar, and Tajana Rosing. 2019. SparseHD: Algorithm-hardware co-optimization for efficient high-dimensional computing. In *Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*. IEEE, Los Alamitos, CA, 190–198.

[22] UCI Machine Learning Repository. 1988. Iris Data Set. Retrieved February 17, 2022 from https://archive.ics.uci.edu/ml/datasets/iris.

[23] UCI Machine Learning Repository. 1994. ISOLET Data Set. Retrieved February 17, 2022 from http://archive.ics.uci.edu/ml/datasets/ISOLET.

[24] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935.

[25] Pentti Kanerva. 2009. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation* 1, 2 (2009), 139–159.

[26] Geethan Karunaratne, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. 2020. In-memory hyperdimensional computing. *Nature Electronics* 3, 6 (2020), 327–337.

[27] Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing. 2020. GenieHD: Efficient DNA pattern matching accelerator using hyperdimensional computing. In *Proceedings of the 2020 Design, Automation, and Test in Europe Conference and Exhibition (DATE'20)*. IEEE, Los Alamitos, CA, 115–120.

[28] Yeseong Kim, Mohsen Imani, and Tajana S. Rosing. 2018. Efficient human activity recognition using hyperdimensional computing. In *Proceedings of the 8th International Conference on the Internet of Things*. 1–6.

[29] Gunjae Koo, Kiran Kumar Matam, I. Te, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading communication with computing near storage. In *Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. IEEE, Los Alamitos, CA, 219–231.

[30] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 137–152.

[31] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-Mei Hwu. 2019. DeepStore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 224–238.

[32] Peer Neubert, Stefan Schubert, and Peter Protzel. 2019. An introduction to hyperdimensional computing for robotics. *KI-Künstliche Intelligenz* 33, 4 (2019), 319–330.

[33] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[34] Abbas Rahimi, Simone Benatti, Pentti Kanerva, Luca Benini, and Jan M. Rabaey. 2016. Hyperdimensional biosignal processing: A case study for EMG-based hand gesture recognition. In *Proceedings of the 2016 IEEE International Conference on Rebooting Computing (ICRC'16)*. IEEE, Los Alamitos, CA, 1–8.

[35] Abbas Rahimi, Pentti Kanerva, José del R. Millán, and Jan M. Rabaey. 2017. Hyperdimensional computing for noninvasive brain-computer interfaces: Blind and one-shot classification of EEG error-related potentials. In *Proceedings of the 10th EAI International Conference on Bio-inspired Information and Communications Technologies*.

[36] Abbas Rahimi, Pentti Kanerva, and Jan M. Rabaey. 2016. A robust and energy-efficient classifier using brain-inspired hyperdimensional computing. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, New York, NY, 64–69.

[37] Abbas Rahimi, Pentti Kanerva, and Jan M. Rabaey. 2016. A robust and energy-efficient classifier using brain-inspired hyperdimensional computing. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, New York, NY, 64–69.

[38] O. Rasanen and J. Saarinen. 2016. Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns. *IEEE Transactions on Neural Networks and Learning Systems* 27, 9 (2016), 1878–1889. https://doi.org/10.1109/TNNLS.2015.2462721

[39] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *Proceedings of the 2019 USENIX Annual Technical Conference*. 379–394.

[40] Mohammad Samragh, Mohammad Ghasemzadeh, and Farinaz Koushanfar. 2017. Customizing neural networks for efficient FPGA implementation. In *Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. IEEE, Los Alamitos, CA, 85–92.

[41] Manuel Schmuck, Luca Benini, and Abbas Rahimi. 2019. Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory. *ACM Journal on Emerging Technologies in Computing Systems* 15, 4 (2019), 1–25.

[42] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. 67–80.

[43] Michael C. Thrun and Alfred Ultsch. 2020. Clustering benchmark datasets exploiting the fundamental clustering problems. *Data in Brief* 30 (2020), 105501.

[44] UCI Machine Learning Repository. 2012. Daily and Sports Activities Data Set. Retrieved February 17, 2022 from https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities.

[45] Tony F. Wu, Haitong Li, Ping-Chen Huang, Abbas Rahimi, Jan M. Rabaey, H.-S. Philip Wong, Max M. Shulaker, and Subhasish Mitra. 2018. Brain-inspired computing exploiting carbon nanotube FETs and resistive RAM: Hyperdimensional computing case study. In *Proceedings of the 2018 IEEE International Solid-State Circuits Conference (ISSCC'18)*. IEEE, Los Alamitos, CA, 492–494.

Received February 2021; revised September 2021; accepted December 2021