THRIFTY: Training with Hyperdimensional Computing across Flash Hierarchy

Saransh Gupta
sgupta@ucsd.edu
University of California, San Diego

Justin Morris
justinmorris@ucsd.edu
University of California, San Diego
and San Diego State University

Mohsen Imani
m.imani@uci.edu
University of California, San Diego
and University of California, Irvine

Ranganathan Ramkumar
rramkuma@ucsd.edu
University of California, San Diego

Jeffrey Yu
jey070@ucsd.edu
University of California, San Diego

Aniket Tiwari
artiwari@ucsd.edu
University of California, San Diego

Baris Aksanli
baksanli@sdsu.edu
San Diego State University

TajanaŠimuni ćRosing
tajana@ucsd.edu
University of California, San Diego

ABSTRACT

Hyperdimensional computing (HDC) is a brain-inspired computing paradigm that works with high-dimensional vectors, *hypervectors*, instead of numbers. HDC replaces several complex learning computations with bitwise and simpler arithmetic operations, resulting in a faster and more energy-efficient learning algorithm. However, it comes at the cost of an increased amount of data to process due to mapping the data into high-dimensional space. While some datasets may nearlyfi t in the memory, the resulting hypervectors more often than not can't be stored in memory, resulting in long data transfers from storage. In this paper, we propose THRIFTY, an in-storage computing (ISC) solution that performs HDC encoding and training across thefl ash hierarchy. To hide the latency of training and enable efficient computation, we introduce the concept of *batching* in HDC. It allows us to split HDC training into sub-components and process them independently. We also present, for thefi rst time, on-chip acceleration for HDC which uses simple low-power digital circuits to implement HDC encoding in Flash planes. This enables us to explore high internal parallelism provided by thefl ash hierarchy and encode multiple data points in parallel with negligible latency overhead. THRIFTY also implements a single top-level FPGA accelerator, which further processes the data obtained from the chips. We exploit the state-of-the-art INSIDER ISC infrastructure to implement the top-level accelerator and provide software support to THRIFTY. THRIFTY runs HDC training completely in storage while almost entirely hiding the latency of computation. Our evaluation overfi ve popular classification datasets shows that THRIFTY is on average 1612× faster than a CPU-server and 14.4× faster than the state-of-the-art ISC solution, INSIDER for HDC encoding and training.

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-6654-2324-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415723

CCS CONCEPTS

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

KEYWORDS

hyperdimensional computing, in-storage computing, classification

ACM Reference Format:
Saransh Gupta, Justin Morris, Mohsen Imani, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Baris Aksanli, and TajanaŠimuni ć Rosing. 2020. THRIFTY: Training with Hyperdimensional Computing across Flash Hierarchy. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415723

1 INTRODUCTION

Brain-inspired Hyperdimensional Computing (HDC) is a computation paradigm which represents data in terms of extremely large vectors, called *hypervectors*. These hypervectors may have 10s of thousands of dimensions and present data in the form of a pattern of signals instead of numbers. By representing data in high-dimension 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 [1]. Prior work has shown the suitability of HDC for various applications like activity recognition, face detection, language recognition, image classification, etc [2–4].

While 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 befi t into the memory, eventually killing the process. Recent work has introduced computing capabilities to solid-state disks (SSDs) to process data in storage [5–8]. 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 a SSD or limit the gains by using complex power-hungry accelerators down the storage hierarchy [9]. Such architectures are not able to fully leverage its hierarchical design.

In this paper, 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 which performs HDC encoding and training 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.
- THRIFTY utilizes die-level accelerators to convert raw data into hypervectors locally in all the flash planes in parallel. Unlike previous work, our accelerator is simpler and hides its computation latency by the long read times of raw data from flash arrays.
- 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 present primitives to enable the FPGA to seamlessly work with the die-level accelerators.

We evaluate THRIFTY over five popular classification datasets for HDC encoding and training. Our experimental results show that THRIFTY is on average 1612× faster than a CPU-server and 14.4× faster than the state-of-the-art ISC solution, INSIDER.

2 RELATED WORK

Hyperdimensional Computing: Prior work applied the idea of hyperdimensional computing to a wide range of learning applications, including language recognition [10], speech recognition [11], gesture detection [12], human-brain interaction [13], and sensor fusion prediction [2]. Prior work also tried to design different hardware accelerators for HD computing. This include accelerating HD computing on existing FPGA, ASIC, and processing in-memory platforms [14–16]. However, these solutions do not scale well with the number of classes and dimensions, primarily due to the data movement issue. In contrast, our proposed THRIFTY accelerates the entire encoding and training phases of HD computing by fundamentally addressing data movement and memory requirement issues. In addition, THRIFTY scales with the size of data and the complexity of 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 [17]. To alleviate these issues prior work introduced ISC architectures [7, 18]. These work exploit the embedded cores present in the SSD controller to implement ISC. Another set of work in [5, 6, 9] used ASIC accelerators in SSD for specific workloads. The work in [8] proposed a full-stack storage system to reduce the host-side I/O stack latency. However, all these works propose single-level computing in storage. THRIFTY on the other hand, is the first work to push the computing all the way down to the flash die to extract maximum parallel. It also uses a top level accelerator to provide addition layer of computing. The combination provide a faster implementation while consuming minimal power.

3 HYPERDIMENSIONAL COMPUTING

Brain-inspired Hyperdimensional (HDC) computing has been proposed as the alternative computing method that processes the cognitive tasks in a more light-weight way [1, 10]. HDC offers an efficient learning strategy without overcomplex 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 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 [3, 4]. 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 $\mathbf{PM}$ 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 be 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

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

sample in i^{th} class, each class hypervector is trained as follows:

$$C^i = \sum_j H^i_j = \langle c^i_D, \cdots, c^i_1 \rangle \quad (2)$$

3.3 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, $\{C_1, C_2, \cdots, C_k\}$. We measure the similarity between a query and a 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 similarities 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.

3.4 Challenges

HDC is light-weight enough to run at acceptable speed on a CPU [19]. Utilizing a parallel architecture can significantly speed up the execution time of HDC [16]. 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 in order 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, in-storage computing (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 [8]. The state-of-the-art application specific ISC [9] try to exploit the internal SSD bandwidth but provide only one-level of computing, which fails to accelerate applications which either (i) have a computing logic that is too complex to implement using the small accelerator or (ii) require post-processing computation steps. THRIFTY aims to overcome these issues by breaking complex HDC training algorithms into simpler, both data-size and computation-wise, parallelizable tasks. Then, THRIFTY 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 THRIFTY DESIGN

THRIFTY is an ISC design that performs HDC encoding and training completely in storage. Figure 1 shows an overview of THRIFTY SSD architecture. A flash die consists of multiple flash planes, each of which generates a page during a read cycle. THRIFTY inserts a simple low-power accelerator, die-level accelerator (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 SSD-level FPGA, which accumulates these hypervectors in batches in the top-level accelerator (green on bottom left in Figure 1). THRIFTY uses a scratchpad (green on 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

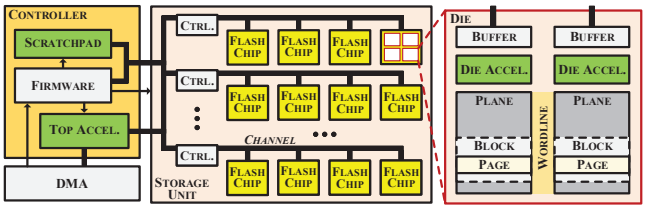

Figure 1: THRIFTY SSD Overview. The components added by THRIFTY are shown in green.

available, without waiting for the remaining data. The top-level accelerator is a FPGA which uses INSIDER acceleration cluster [8] to implement HDC accumulation and other operations. We utilize the INSIDER's software stack to connect THRIFTY to the rest of the system. We modify the SSD drivers and INSIDER virtual files mechanism to enable computing in flash chips and make it visible to the FPGA.

4.1 Batched HDC Training in THRIFTY

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 the encoded hypervectors and then perform HDC training over them. Hence, we employ batching to perform partial training with the 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 spit 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 3.

$$C^i = \sum_{j=1}^{b} H^i_j + \sum_{j=b+1}^{2b} + \ldots + \sum_{j=((s-1)b)+1}^{s} H^i_j \quad (3)$$

Batching allows THRIFTY to process a subset of encoded hypervectors together. THRIFTY 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 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 the top-level FPGA accelerator? Some application may need too many dimensions or have too many classes to store all model

hypervectors at the FPGA, which at best may have few MBs 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, THRIFTY 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 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, THRIFTY 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 one to account for the sign bit. In this case, assuming the original data is represented with 32 bits, THRIFTY sends less data than the data movement required to send the original data in low-dimensional space when $32nb > D((\log_2 4\sqrt{b}) + 1)/32$

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 4-8 flash chips as shown in Figure 1. The flash chip may consist of several flash dies which 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 SSD happen in page granularity where the size of pages usually ranges from 2KB-16KB [20]. 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. THRIFTY plane-accelerator encodes an entire page with raw data to generate a D dimensional hypervector. Let us assume the SSD page size to be 4KB (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 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 the 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 1024 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, THRIFTY 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. THRIFTY writes the output of the accelerator to the page buffer of the plane, which serves as the response to the original SSD read request.

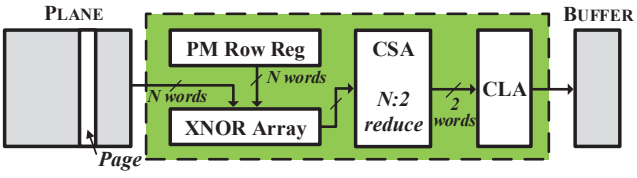

Figure 2: THRIFTY die accelerator

4.2.2 Storing Input Data. The accelerator above 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 size is too small or just larger than the page size. For example, in a page-aligned feature vector setting, a 4KB page may fit only one 512B feature instead of eight. Also, a 5KB feature vector may occupy two complete pages. To alleviate the issue, we propose a cross-plane storing scheme, which considers all the 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:

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

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 $d \in \{0, 1, ...n_p\}$. 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

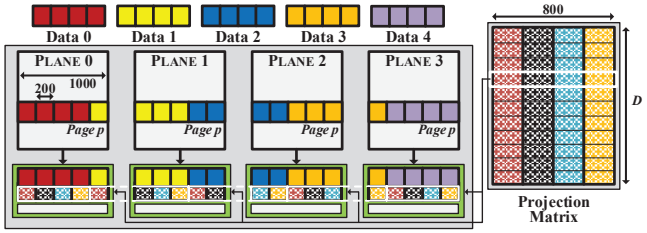

Figure 3: Data storage scheme in THRIFTY and the corresponding segmentation of the projection matrix. Data represents a feature vector.

the right. This allows THRIFTY 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, THRIFTY uses the same segmentation size. However, the number of segments in a page are given by:

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

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 Accelerator with New Data Storing Scheme. While 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 THRIFTY.

As detailed in Section 4.1, a set of encoded hypervectors can be added dimension-wise without interfering with HDC training process as long as they belong to the same final trained hypervector, for example 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), i.e.,

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

Now, to add multiple FVs together, we just need to make sure that an element in a FV is being multiplied with the corresponding weight of the PM. 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 these hypervectors will be added together during training. Hence, the final hypervector will contain all the information.

To support this strategy in THRIFTY 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. So, 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.

4.3 Training at top-level

The encoded hypervectors from flash chips are used for training in the top-level accelerator, which is implemented on an FPGA present in the SSD. We use FPGA because it is flexible with the application parameters and can be configured using the primitives provided by INSIDER [8]. The encoded hypervectors come with class labels. During training, they are accumulated into the corresponding class (or model) hypervectors. At the end of training we obtain an output hypervector for each class that in turn represents all the input samples belonging to that class.

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 class label of an incoming hypervector is used to index it to the corresponding class input queue. The size of the queue is determined based on the frequency of the inputs, the number of classes, and dimensionality D. 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 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. While the computation can also be fully parallelized over all dimension, the large size of hypervectors and the limited read ports of the memory make it impractical. Hence, we divide the hypervectors into partitions to allow partial parallelism. The final class hypervectors are sent to the host.

If an application has too many classes or requires extremely large number of dimensions, then the FPGA may not have enough space to store all the class hypervectors. In such a case, we allocate the memory for the maximum number of class hypervectors, C_{max}. We assign labels to these classes with respect to the incoming hypervectors. Hence, the first set of incoming hypervectors belonging to C_{max} different classes are processed as before. We introduce an addition queue that indexes, along with their labels, the incoming hypervectors not associated with any of the active C_{max} class. Whenever the queue is full, one of the C_{max} class hypervectors is sent to the CPU host. The corresponding memory is allocated for the class to which the first hypervector in the queue belongs. The class hypervector sent to the host is the one that has accumulated the most incoming hypervectors.

4.4 Software Support

THRIFTY derives its base system-architecture from INSIDER [8]. The INSIDER framework is an API which, 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 which 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 doesn't care about the source and/or destination of the data.

4.4.1 THRIFTY Host-Side Support. INSIDER API uses POSIX-like I/O functionaly 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 THRIFTY Drive-Side Architecture. THRIFTY implements its top-level accelerator described in Section 4.3 as an INSIDER acceleration cluster, which enables the final training step. However, INSIDER system doesn't support THRIFTY's die-level acceleration because the standard read/write drive operations can't readily accommodate on-the-fly change in data size while reading encoded pages and writing projection matrix elements to the on-die accelerator.

THRIFTY introduces the processing capability between flash planes and page buffers but sometimes only raw data may be required. Hence, THRIFTY 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 4. 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 while a compute read responds with multiple but fixed number of pages.

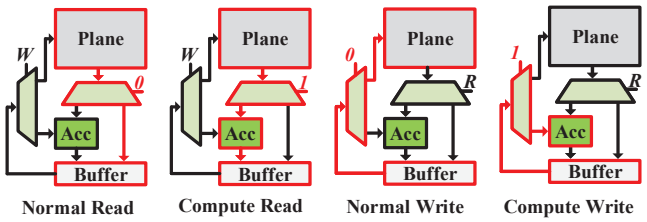

Figure 4: Different read and write modes in THRIFTY. The components in red are active during the operation.

THRIFTY 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.

THRIFTY 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 4. 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 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. While sequential, this step has negligible latency overhead because it can be performed in parallel for all 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 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.

5 RESULTS

5.1 Experimental Setup

We develope a simulator for THRIFTY which supports parallel read and write accesses to the flash chips. We utilize Verilog and Synopsys *Design Compiler* to implement and synthesize our die-level accelerator at 45nm and scale it down to 22nm. The top-level FPGA accelerator has been synthesized and simulated in Xilinx Vivado. For THRIFTY drive simulation, we assume the characteristics similar to 1TB Intel DC P4500 PCIe-3.1 SSD connected to an Intel(R)

Table 1: THRIFTY 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	$53us$	**Pages/Block**	128
FPGA	$XCKU025$	**Scratchpad Size**	$4MB$
Avg Power/DA	$8mW$	**DA Latency**	$1.02ns$

*DA: Die-accelerator

Xeon(R) CPU E5-2640 v3 host. The parameters for THRIFTY are shown in Table 1.

We compare THRIFTY with 7th Gen 2.4GHz Kaby Lake Intel Core i5 CPU with 8MB RAM and 256 GB SSD. We also compare it with a 3.5GHz Intel(R) Xeon(R) CPU E5-2640 v3 CPU server with 256GB RAM and 2TB local disk. We also compare THRIFTY with INSIDER [8] and DeepStore [9], 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 which employs specialized accelerators in SSD. For all our experiments, including those for other ISC solutions, the data is assumed to be channel-striped and stored using THRIFTY's proposed scheme.

5.2 Workloads

We evaluate the efficiency of THRIFTY on five popular classification applications, as listed below:

Speech Recognition (`ISOLET`): The goal is to recognize voice audio of the 26 letters of the English alphabet [21].

Face Recognition (`FACE`): We exploit Caltech dataset of 10,000 web faces [22]. Negative training images, i.e., non-face images, are selected from CIFAR-100 and Pascal VOS 2012 datasets [23].

Activity Recognition (`UCIHAR`): The dataset includes signals collected from motion sensors for 8 subjects performing 19 different activities [24].

Medical Diagnosis (`CARDIO`): This dataset provides medical diagnosis based on cardiotocography information about each patient [25].

Gesture Recognition (`EMG`): This dataset contains EMG readings for five different hand gestures [26].

5.3 Comparison with CPU and CPU Server

We first compare THRIFTY with CPU and CPU-based server running state-of-the-art implementations of HDC encoding and training 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. The runtime for different platforms is shown in Figure 5. We observe that THRIFTY is on average 3405× and 1612× faster than CPU and CPU-server, respectively. Our evaluations show that the performance of THRIFTY increases linearly with an increase in the dataset size. This happens because more data samples result in more huge hypervectors to generate

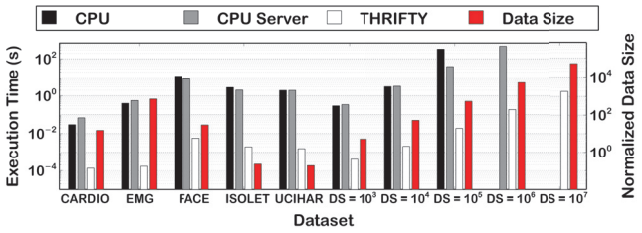

Figure 5: Runtime comparison of HDC encoding and training in THRIFTY with other platforms. The bars in red shows the size of raw data normalized to the total size of corresponding class hypervectors in THRIFTY.

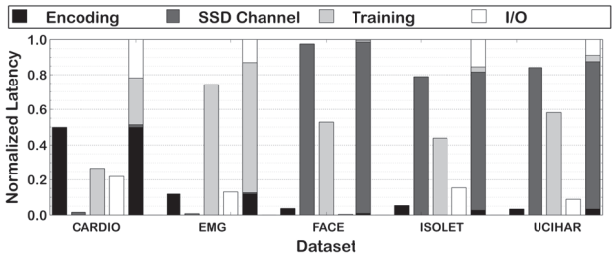

Figure 6: Breakdown of latency of different stages of HDC normalized to the total latency.

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 THRIFTY 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.

Figure 5 also shows the size of raw input data in each case normalized to the size of the corresponding trained class hypervectors. While THRIFTY only sends class hypervectors from drive to the host, 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.

5.4 THRIFTY Efficiency

Figure 6 shows the breakdown of THRIFTY 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 THRIFTY 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 example of `FACE` dataset, even though the training takes

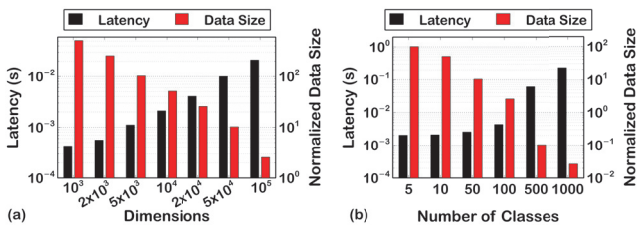

Figure 7: Change in HDC runtime and raw data size to hypervector ratio with (a) dimensions and (b) number of classes.

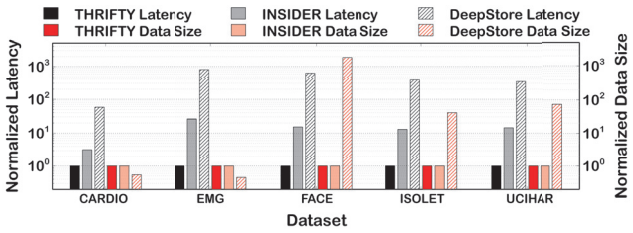

Figure 8: Runtime and data transfer size comparison of THRIFTY with INSIDER [8] and DeepStore [9]

more than half of the total latency, a negligible portion of it actually contributes to the overall latency. It shows that THRIFTY stages are able to hide some of their latency.

To demonstrate the scalability provided by THRIFTY, 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 7a shows that the latency of THRIFTY increases linearly with an increase in the number of dimensions, showing that THRIFTY 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 class, while keeping its size fixed to 10^4 samples and D as 10^3. Figure 7b shows that the THRIFTY latency has minor changes with the number of classes when we have less 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 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 7b. In addition to the time spent in training, transferring the class hypervectors to host creates a major bottleneck. This is also evident from the data size ratio which declines for 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 Comparison with Existing Solutions

We compare the performance and data transfer efficiency of THRIFTY with INSIDER [8] and DeepStore [9]. In our experiments, INSIDER performs both encoding and training 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 THRIFTY die-level accelerator. During ISC, Deepstore encodes the raw data into hypervectors and sends those hypervectors to the host for training. Figure 8 shows the change in latency and data transfer size for the three ISC solutions. We observe that THRIFTY is on an average 14.4× and 446.8× faster than INSIDER and DeepStore, respectively. While encoding in DeepStore takes the same time as THRIFTY, transferring hypervector from SSD to host and further training on them on CPU increases the execution time of DeepStore significantly. On the other hand, the SSD channel bottleneck faced by THRIFTY is relaxed in case of INSIDER since it only transfers raw data. However, the FPGA-based HDC encoding+training are on an average 21× slower as compared to FPGA-based training. Also, since INSIDER performs training in SSD, it transfers the same amount of data to the host as THRIFTY. However, by transferring untrained hypervectors, DeepStore increases the amount of data transferred on an average by 397× as compared to THRIFTY.

6 CONCLUSION

In this paper, 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 encoding and training in-storage over multiple data samples in parallel. We proposed batched HDC training to enable partial processing of HDC hypervectors. We further proposed die-level and top-level accelerators for HDC encoding and training respectively. Our evaluation shows that THRIFTY is on average 1612× faster than a CPU-server and 14.4× faster than INSIDER for HDC encoding and training.

ACKNOWLEDGMENTS

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 grant, and also NSF grants # 1527034, #1730158, #1826967, and #1911095.

REFERENCES

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

[2] O. Rasanen and J. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–12, 2015.

[3] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[4] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, ACM, 2016.

[5] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[6] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, *et al.*, "Biscuit: A framework for near-data processing of big data workloads," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 153–165, 2016.

[7] G. Koo, K. K. Matam, I. Te, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 219–231, IEEE, 2017.

[8] Z. Ruan, T. He, and J. Cong, "Insider: designing in-storage computing system for emerging high-performance drive," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, pp. 379–394, 2019.

[9] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. De Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "Deepstore: In-storage acceleration for intelligent queries," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 224–238, 2019.

[10] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, ACM, 2016.

[11] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, pp. 1–8, IEEE, 2017.

[12] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey, "Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2016.

[13] A. Rahimi, P. Kanerva, J. d. R. Millán, and J. M. Rabaey, "Hyperdimensional computing for noninvasive brain-computer interfaces: Blind and one-shot classification of eeg error-related potentials," in *10th EAI Int. Conf. on Bio-inspired Information and Communications Technologies*, no. CONF, 2017.

[14] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–25, 2019.

[15] T. F. Wu *et al.*, "Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study," in *ISSCC*, pp. 492–494, IEEE, 2018.

[16] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *IEEE HPCA*, IEEE, 2017.

[17] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 137–152, 2017.

[18] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: a user-programmable ssd," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pp. 67–80, 2014.

[19] M. Imani *et al.*, "A binary learning framework for hyperdimensional computing," in *DATE*, 2019.

[20] W. Cheong, C. Yoon, S. Woo, K. Han, D. Kim, C. Lee, Y. Choi, S. Kim, D. Kang, G. Yu, *et al.*, "A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time," in *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 338–340, IEEE, 2018.

[21] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[22] G. Griffin, A. Holub, and P. Perona, "Caltech-256 object category dataset," 2007.

[23] M. Everingham, S. A. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International journal of computer vision*, vol. 111, no. 1, pp. 98–136, 2015.

[24] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine," in *International workshop on ambient assisted living*, pp. 216–223, Springer, 2012.

[25] "Cardiotocography data set." https://archive.ics.uci.edu/ml/datasets/cardiotocography.

[26] S. Benatti, E. Farella, E. Gruppioni, and L. Benini, "Analysis of robust implementation of an emg pattern recognition based control.," in *BIOSIGNALS*, pp. 45–54, 2014.