Revisiting HyperDimensional Learning for FPGA and Low-Power Architectures

Mohsen Imani[†⋆], Zhuowen Zou[*], Samuel Bosch[‡], Sanjay Anantha Rao[*], Sahand Salamat[*]
Venkatesh Kumar[*], Yeseong Kim[Ψ], Tajana Rosing[*]
[†]Department of Computer Science, University of California Irvine
[*]Department of Computer Science and Engineering, UC San Diego
[‡] Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology
[Ψ]Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology
[⋆] Email: m.imani@uci.edu

***Abstract*—Today's applications are using machine learning algorithms to analyze the data collected from a swarm of devices on the Internet of Things (IoT). However, most existing learning algorithms are overcomplex to enable real-time learning on IoT devices with limited resources and computing power. Recently, Hyperdimensional computing (HDC) is introduced as an alternative computing paradigm for enabling efficient and robust learning. HDC emulates the cognitive task by representing the values as patterns of neural activity in high-dimensional space. HDC first encodes all data points to high-dimensional vectors. It then efficiently performs the learning task using a well-defined set of operations. Existing HDC solutions have two main issues that hinder their deployments on low-power embedded devices: (i) the encoding module is costly, dominating 80% of the entire training performance, (ii) the HDC model size and the computation cost grow significantly with the number of classes in online inference.**

In this paper, we proposed a novel architecture, LookHD, which enables real-time HDC learning on low-power edge devices. LookHD exploits computation reuse to memorize the encoding module and simplify its computation with single memory access. LookHD also address the inference scalability by exploiting HDC governing mathematics that compresses the HDC trained model into a single hypervector. We present how the proposed architecture can be implemented on the existing low power architectures: ARM processor and FPGA design. We evaluate the efficiency of the proposed approach on a wide range of practical classification problems such as activity recognition, face recognition, and speech recognition. Our evaluations show that LookHD can achieve, on average, 28.3× faster and 97.4× more energy-efficient training as compared to the state-of-the-art HDC implemented on the FPGA. Similarly, in the inference, LookHD is 2.2× faster, 4.1× more energy-efficient, and has 6.3× smaller model size than the same state-of-the-art algorithms.

***Index Terms*—HyprDimensional computing, Brain-inspired computing, Machine learning, Real-time learning, FPGA**

I. Introduction

With the emergence of the Internet of Things (IoT), many applications run machine learning algorithms to perform cognitive tasks. The learning algorithms have been shown effectiveness for many tasks, e.g., object tracking [1], speech recognition [2], [3], image classification [4], [5], etc. However, the high computational complexity and memory requirement of existing deep learning algorithms hinder usability to a wide variety of real-life embedded applications where the device resources and power budget is limited [6], [7], [8], [9]. Therefore, we need alternative learning methods that can train on the less-powerful IoT devices, while providing good enough classification accuracy.

To achieve real-time performance with high energy efficiency, we need to rethink not only how we accelerate machine learning algorithms in hardware, but also to redesign the algorithms themselves using strategies that more closely model the ultimate efficient learning machine: *the human brain*. Hyperdimensional computing (HDC) is one such strategy developed by interdisciplinary research [10]. It is based on a short-term human memory model, Sparse distributed memory, emerged from theoretical neuroscience [11]. HDC is motivated by the understanding that the human brain operates on *high-dimensional* representations of data originating from the large size of brain circuits [12]. It thereby models the human memory using points of a high-dimensional space, that is, with *hypervectors*. The hyperspace typically refers to tens of thousand dimensions. HDC incorporates learning capability along with typical memory functions of storing/loading information. It mimics several essential functionalities of the human memory model with vector operations, which are computationally tractable and mathematically rigorous in describing human cognition.

HDC is well suited to address learning tasks for IoT systems as: (i) HDC models are computationally efficient (highly parallel at heart) to train and amenable to hardware level optimization [13], (ii) HDC offers an intuitive and human-interpretable model [14], [15], [16], (iii) it offers a computational paradigm that can be applied to cognitive as well as learning problems [14], [17], [18], [19], [20], (iv) it provides strong robustness to noise – a key strength for IoT systems, and (v) HDC can naturally enable secure and light-weight learning. These features make HDC a promising solution for today's embedded devices with limited storage, battery, and resources, as well as future computing systems in deep nano-scaled technology, where devices may have high noise and variability [13], [21], [22], [23].

Recently, several companies started exploiting the HDC capability to enable general intelligence in IoT devices, including WebFeet [24], Vicarious [25], Numenta [26], [27], IBM [28], and Google [29]. Most existing researches are focused on exploiting the HDC robustness to design approximate analog accelerator [21], [22], [23]. However, to make the HDC practical, we need to exploit the HDC robustness on the existing embedded platforms, e.g., CPU and FPGA. As prior work showed, the existing digital platforms can get limited benefit from hardware approximation [30], [31], [32]. The key motivation of our work is to significantly speedup the HDC learning on embedded platforms by redesigning algorithm-hardware that accelerates crucial HDC functionalities.

HDC performs the learning task after encoding all data points to high-dimensional space. This encoding requires to compute thousands (e.g., 10,000) operations for each element of data in the original domain, e.g., performing permutation (rotational shift) and addition of randomly generated bipolar/binary hypervectors [10], [33], [34]. This makes the encoding computationally expensive. Our experiments on several practical applications show that the encoding takes about 80% of the total training's execution time. During inference, HDC uses the same encoding module to map a test data point to high-dimensional space (query hypervector). Then, HDC checks the similarity of the query hypervector with all pre-trained class hypervectors. This similarity check takes a major cost of the HDC during the inference, i.e., 83% average performance for all tested applications, as it involves many multiplications that grow with the number of classes [21], [35]. This degrades the scalability of the HDC model and increases the computation cost of applications with many classes.

In this paper, we propose LookHD, an architectural solution that accelerates HDC in both training and inference, making it significantly efficient for today's embedded processors. LookHD exploits computational reuse to eliminate the costly encoding from the HDC training while addressing the scalability issue of the HDC inference by compressing the model size and reducing the computation costs. The followings summarize the main contributions of the paper:

- We propose a lookup-based approach that eliminates the costly encoding operations from the HDC. LookHD pre-stores all possible encoded values and replaces the costly encoding module with single memory access. Unlike existing HDC algorithms [14], [33], [36] that needs to combine all the encoded hypervectors, LookHD only counts the number of patterns repeated in the pre-stored hypervectors and generates the hypervector model once at the end of the training. This results in eliminating the encoding module, significantly accelerating the HDC training.
- LookHD addresses the scalability issue of the inference phase due to the model size. In contrast to conventional HDC approaches that use multiple hypervectors corresponding to each class, LookHD compresses them into a single hypervector. The combined hypervector stores the information of all classes in significantly lower-sized memory, which makes it suitable for embedded devices with limited resources. This approach also significantly accelerates the inference by reducing the number of computations, e.g., multiplications.
- We present two architecture options to implement LookHD on embedded devices (with less than 10W power budget). Along with a software framework design for low-power ARM processors, we propose an FPGA implementation that performs both training and inference. The FPGA design utilizes a fully pipelined structure in order to maximize the throughput and FPGA resource utilization. All proposed optimizations are general and can be implemented on any digital processor, including an ASIC chip.

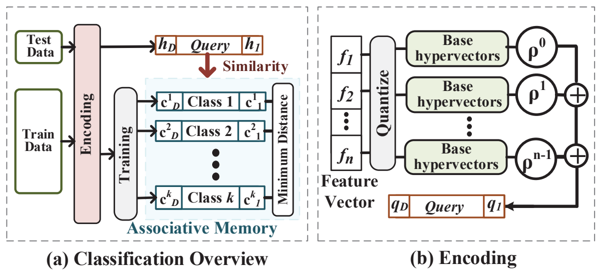


Fig. 1. (a) HDC classification Overview, (b) HDC encoding functionality.

We evaluate the efficiency of the proposed approach on a wide range of practical learning problems, including speech recognition, activity recognition, and face recognition. FPGA-based LookHD is implemented on Kintex-7 FPGA KC705. Our evaluation shows that LookHD achieves, on average 28.3× faster and 97.4× higher energy efficient training, as compared to the state-of-the-art HDC solution [37], [38] implemented on the FPGA. For the inference tasks, LookHD is 2.2× faster, 4.1× more energy-efficient, and has 6.3× smaller model size.

II. HDC: Functionality & Challenges

Figure 1a shows an overview of HDC performing the classification task on high-dimensional space. The first step of HDC is to map/encode data points from original into high-dimensional space. The encoded hypervectors are combined during training in order to create a single hypervector representing each class. In the inference, a test data is encoded to high-dimensional space using the same encoding module used for training. The classification is performed by finding a pre-stored class hypervector, which has the highest similarity with the test hypervector. In the following, we explain the details of the HDC functionality.

A. HD Encoding

Figure 1b shows an overview of the HDC encoding module. Let us assume a feature vector $\mathbf{F} = \{f_1,\ f_2, \ldots,\ f_n\}$, with n features ($f_i \in \mathbb{N}$). The goal of encoding is to map a feature vector to a high-dimensional vector, $\mathbf{H} = \{h_1,\ h_2, \ldots,\ h_D\}$ with D dimensions ($h_i \in \mathbb{N}$), where D is in order of thousands, e.g., 10,000 [21], [33], [34]. The encoding keeps the main information of original data as a pattern of values in high-dimensional space. HDC represents the feature values as patterns of bitstreams in HDC space and combines them to preserve the position of each pattern [34].

Alphabets Generation: Instead of representing the features using their value, HDC represents them using a set of hypervectors where their patterns determine their values. First, we find the maximum and minimum feature values, $\{f_{min}, f_{max}\}$, and then quantize that range into q discrete levels, $\{f^q_1, f^q_2, \cdots, f^q_q\}$. Then, we assign a single hypervector to each quantized level $\{\mathbf{L}_1, \mathbf{L}_2, ..., \mathbf{L}_q\}$, where L_1, and L_q are corresponding to f_{min}

and f_{max} values respectively. The level hypervectors are bipolar with D dimensions, $\mathbf{L_i} \in \{-1,1\}^D$. The level hypervectors need to preserve the same similarity distance as the quantized values in the original space. To this end, we randomly generate the first level hypervector (representing f_{min}). The other level hypervectors are generated by filliping D/q random dimensions of the previous level hypervector. Using this method, the $\mathbf{L}_q$ hypervector corresponding to f_{max} will be nearly orthogonal to $\mathbf{L}_1$, while the neighbor levels will have high pattern similarity.

Preserve Position: In the feature vector, the information is stored as a pattern of features in different indices. The encoding needs to keep the information of both feature values and their corresponding position. HD preserves each index position by assigning a fixed permutation to it. Based on HDC theory [10], any permutation of a random hypervector will be nearly orthogonal with the original hypervector:

$$\delta\langle \mathbf{L}, \rho^{(i)}\mathbf{L}\rangle \simeq 0 \qquad (0 < i \leqslant n)$$

where the similarity metric, δ, is a cosine between the two hypervectors, and $\rho^{(i)}L$ is the i-bit rotational shift of $\mathbf{L}$. The orthogonality of a hypervector and its permutation (i.e. circular bitwise rotation) is ensured as long as the hypervector dimensionality is large enough compared to the number of features in the original data point ($D >> n$). As Figure 1 shows, the aggregation of the n binary hypervectors is computed as follows:

$$\mathbf{H} \ = \ \overline{\mathbf{L}}_1 \ + \ \ \rho\ \overline{\mathbf{L}}_2 + \ldots \ + \ \rho^{(n-1)}\overline{\mathbf{L}}_n. \qquad (1)$$

where, $\mathbf{H}$ is the (non-binary) aggregation and $\overline{\mathbf{L}}_i$ is the (binary) hypervector corresponding to the i-th feature of vector F.

B. HD Initial Training & Retraining

In HDC, the training is performed by the element-wise addition of all encoded hypervectors in each existing class. The result of training will be k hypervectors with D dimensions, where k is the number of classes. For example, i^{th} class hypervector can be computed as: $\mathbf{C_i} = \sum_{\forall j \in class_i} \mathbf{H_j}$ After the initial training, HDC revisit the trained model for a few iterations. We call this iterative process as *retraining*. During a single iteration of the retraining, HDC checks the similarity of all training data points, say $\mathbf{H}$, with the trained model. If a data is wrongly classified by the model, HDC updates the model by (i) adding the data hypervector to a class that it belongs to ($\widetilde{\mathbf{C}}^{correct} = \mathbf{C}^{correct} + \mathbf{H}$), and (ii) subtracting it from a class which it is wrongly matched with ($\widetilde{\mathbf{C}}^{wrong} = \mathbf{C}^{wrong} - \mathbf{H}$). The retraining needs to be continued for a few iterations until the HDC accuracy stabilized over the validation data, which is a part of the training dataset.

C. HD Inference

In the inference, HDC uses the same encoding module as the training module to map a test data point to a *query hypervector*. In HDC space, the classification task is performed by checking the similarity of the query with all class hypervectors. Each data point is assigned to a class that has the highest similarity with it. Since HDC information is stored as the pattern of non-binary values, the cosine is suitable for similarity check.

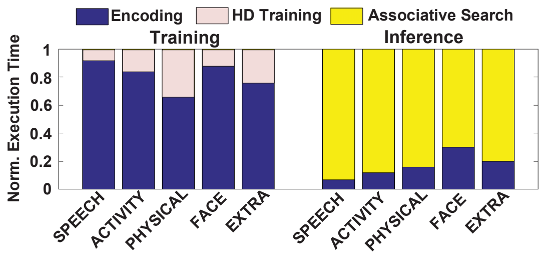


Fig. 2. Breakdown of the encoding, training and associative search execution time during training and inference.

D. HDC Challenges

Training Challenges: As a light-weight classifier, HDC often needs to be trained and tested on embedded devices with limited resources. However, we observe that HDC can be computationally costly when it processes the practical classification applications. Figure 2 shows the normalized execution time of HDC during training and inference on five practical applications including: speech recognition (SPEECH) [39], activity recognition (ACTIVITY) [40], physical monitoring (PHYSICAL) [41], face recognition (FACE) [42], and position recognition using extra sensory (EXTRA) [43]. All evaluations are performed on ARM Cortex A53 CPU using C++ implementation of HDC. The details of each application, such as the number of features (n) and the number of classes (k) are explained in Section VI-A. Our results show that for practical classification applications with large feature sizes, the encoding module dominates the training execution time. For example, for speech recognition with $n = 617$ features, the encoding can take 90% of total training time. This reduces the advantage of HDC as a light-weight classifier. In this paper, we propose a novel approach that significantly reduces the encoding cost. Our design ignores performing the encoding operations by pre-storing all possible encoding results. The details of the proposed approach are explained in Section III.

Inference Challenges: Figure 2 also shows the breakdown of the HDC execution time during the inference. As our results show, in the inference, the associative search takes the majority of the HDC cost. For five tested applications, the associative search takes about 83% of the total inference execution. This is because the cosine similarity involves a large number of multiplications between a query and class hypervectors. In addition, in existing HDC approaches [21], [33], [44], the model size and the computation cost increase linearly with the number of classes. For example, speech recognition with $k = 26$ classes has 13.0× larger model size and 5.2× slower inference computation as compared to face detection with $k = 2$ classes. Since embedded devices often do not have enough memory and computing resources, processing HDC applications with a large number of classes will result in huge computation cost.

In this paper, we propose a novel approach to addresses the HDC inference scalability. Our solution combines all classes into a single hypervector (regardless of the number of classes) by utilizing the orthogonality of high-dimensional vectors. This method not only reduces the HDC model size but also accelerates the inference by removing the majority

TABLE I
THE CHARACTERISTIC OF FIVE PRACTICAL APPLICATIONS PROVIDING MAXIMUM ACCURACY.

Applications	n	q	k	HD Accuracy	Lookup Size (# rows)
Speech Recognition	617	16	26	94.1%	$2^{2,468}$
Activity Recognition	561	8	6	94.6%	$2^{1,683}$
Physical Monitoring	52	8	12	91.3%	2^{156}
Face Recognition	608	16	2	94.1%	$2^{2,432}$
Extra Sensory	225	16	4	70.6%	2^{900}

of multiplication operations from the cosine similarity. In Section IV, we explain the details of the proposed model compression method.

III. LookHD Encoding and Training

In HDC, all training and testing (inference) computations happen on the encoded data in high-dimensional space. Therefore, HDC needs to pay the cost of encoding for all available data points. Practical classification problems are working with data points with hundreds/thousands of features. This significantly increases the cost of the encoding module.

When encoding a data point with n features, as shown in Equation 1, each feature index gets a unique permutation, and each feature value gets one of the possible q quantized levels. Looking at all possible combinations, we can see that the encoded hypervector can get q^n different possibilities. In order to avoid the costly encoding operations, one solution is to pre-store all q^n possible hypervectors in a memory block. This enables computation reuse as we can simplify the encoding operation to a single lookup table search. Here we look at the feasibility of this approach for HDC. Table I shows the number of features for five practical applications. In addition, for each application, Table I lists the minimum number of quantized levels, which results in maximum classification accuracy. Our evaluation shows that in practical applications, the number of features, n, and the number of quantized levels, q, are much higher than a range that can be stored in reasonable memory size. For an example of speech recognition, each feature vector can be encoded to $q^n = 16^{617}$ different possible D-dimensional hypervectors. To make the lookup-based encoding feasible, in this section, we propose LookHD, which significantly reduces the number of features and the number of quantized levels. LookHD reduces the number of feature values by splitting the feature vector into small chunks, where all chunks can be encoded using the same encoding module. LookHD also proposes a novel quantization approach that enables HDC to provide the maximum classification accuracy using much lower quantization. In the following, we explain the details of the proposed approach.

A. Splitting Features

Here, we propose a novel approach which enables HDC to encode a feature vector which has been split into the small chunks. Assume a data point with n features, $\mathbf{F} = \{f_1, f_2, \cdots, f_n\}$, our approach splits the feature vector into m equal sequential chunks, $\mathbf{F} = \{\mathbf{F}^1, \mathbf{F}^2, \ldots, \mathbf{F}^m\}$, where $\mathbf{F}^i = \{f_{i+1},\ f_{i+2}, , f_{i+r}\}$ and $r = n/m$. Instead of encoding all n features at once, our approach encodes each feature chunk individually using the same encoding module. The encoding in the i^{th} chunk is performed as follows:

$$H_i = \overline{\mathbf{L}}_1 + \rho\overline{\mathbf{L}}_2 + \cdots + \rho^{r-1}\overline{\mathbf{L}}_r \qquad (2)$$

where $\overline{\mathbf{L}} \in \{\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_q\}$ are the same quantized hypervectors used in conventional encoding.

To complete the encoding module, we need to combine the encoded hypervectors through all chunks and represent them using a single hypervector. One naive approach is to aggregate the chunk hypervectors by simply adding them together. Although this method keeps the information of all chunks (i.e., the pattern of each individual encoded chunk) in a combined hypervector, it does not preserve the order of combination. In this paper, we exploit the mathematical orthogonality of random vectors in high-dimensional space to combine the chunk hypervectors. To consider the position of each chunk on a combined hypervector, LookHD generates m random bipolar hypervectors, $\{\mathbf{P}_1, \mathbf{P}_2 \ldots, \mathbf{P}_m\}$ with D dimensions ($\mathbf{P}_i \in \{-1,1\}^D$). Since these hypervector are generated randomly, they will have have nearly orthogonal distribution [34]:

$$\delta\langle \mathbf{P}_i,\ \mathbf{P}_j\rangle \simeq 0 \qquad (0 < i, j \leq m,\ \ i \neq j)$$

We preserve the the position of each chunk using:

$$\mathbf{H} = \mathbf{P}_1 * \mathbf{H}_1 + \mathbf{P}_2 * \mathbf{H}_2, \ldots, \mathbf{P}_m * \mathbf{H}_m \qquad (3)$$

where the combined $\mathbf{H}$ hypervector stores the information of all chunks as well as their order of combination. However, this combination is not error-free, since the $\mathbf{P}$ hypervectors are not entirely orthogonal. The combined hypervector may lose information when we store the information of too many chunks. In Section VI-B, we discuss the impact of the number of chunks on the HDC classification accuracy.

B. Quantization Reduction

Although splitting the feature vector reduces the number of pre-stored hypervectors from q^n to q^r, the value of q can still be very large such that it makes the lookup approach infeasible. Looking at the Table I, we can see that applications usually require $q = 8$ or $q = 16$ to provide their highest accuracy. For example, speech recognition provides the maximum accuracy using $q = 16$. Using this quantization level even with tiny chunk size (e.g., $r = 5$), we still require to pre-store $16^5 = 2^{20}$ hypervectors, which is still very large for practical implementation. To further reduce the number of possible encoded hypervectors, we need to reduce the number of quantized levels.

The blue line in Figure 4 shows the impact of reducing the number of quantized levels on speech recognition accuracy.

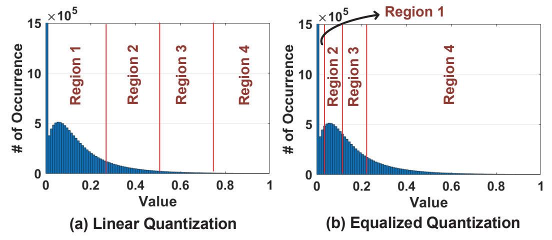


Fig. 3. Feature quantization using linear and the proposed equalized approach.

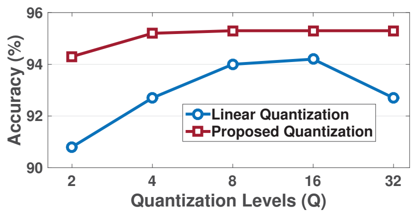


Fig. 4. Impact of linear and equalized quantization on the speech recognition classification accuracy.

The results show that reducing the number of quantized levels to $q = 2$ and $q = 4$ degrades the HDC classification accuracy by 3.4% and 1.5%, respectively. Thus, reducing the q value comes at the expense of a large quality loss. The second approach to reducing the number of possible encoded hypervectors is to use a smaller chunk size ($r <<$). However, this degrades the advantage of the lookup approach by increasing the cost of chunk aggregation.

Here, we propose a novel approach that significantly reduces the number of quantized levels with no negative impact on classification accuracy [45]. Figure 3a shows the distribution of feature values, sampling 5% data points from the speech recognition dataset. The graph indicates that the feature values have non-uniform distribution. Therefore, increasing the number of quantizations linearly results in generating the levels that will be used rarely during the encoding. Instead of linearly quantizing the feature values, our approach selects the quantization boundaries such that all levels get a similar number of values. Figure 3b shows how our approach quantizes speech recognition feature values into $q = 4$ equalized levels. Figure 4 compares the classification accuracy of LookHD using conventional and proposed quantization. In conventional quantization, increasing the number of levels may degrade the classification accuracy, while the proposed quantization approach results in similar or better accuracy using larger q. This is because, in linear quantization, the new levels may be assigned to a range that may not be equally used during the classification, making the classification task more complicated. Our results also show that the proposed quantization results in much higher accuracy than linear quantization. For example, in speech recognition, the proposed quantization to $q = 4$ levels provide 1.2% higher classification accuracy than HDC using $q = 16$ linear quantization.

LookHD learns from highly quantized input data, which is not possible in the original space. HDC encoding is non-linear and projects input data with a small difference to relatively isolated data in sparse high-dimensional space. Due to the high-dimensionality of the HDC encoding module, there are several possibilities that each encoded value can get, thus a small feature difference in original space can be projected to large difference in HDC space.

C. Lookup-based Encoding

Splitting the feature vectors along with equalized quantization reduces the number of possible encoded hypervectors. For an example of speech recognition, splitting the feature vector to $r = 5$ chunk size and using $q = 4$ equalized quantization

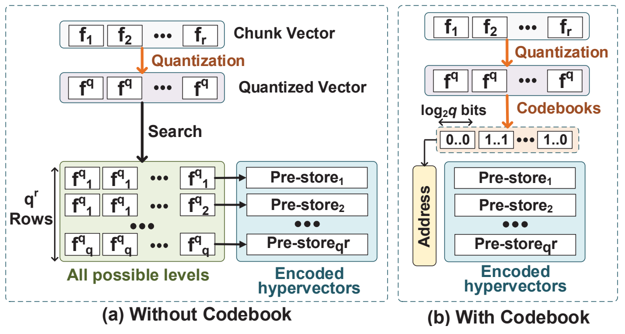


Fig. 5. Encoding of each feature chunk without and with using codebook.

reduces the number of possible encoded hypervectors from 16^{617} to 4^5, while ensuring the same accuracy as the baseline HD. Figure 5 shows the overview of lookup-based encoding in a single chunk. In the first step, we quantize each input features into one of the possible quantization levels. Then, we access encoding results by searching for r quantized features in a lookup table that pre-stored all possible combinations. In hardware, the lookup table is expensive since it involves many compare/search operations.

To avoid costly search operation, LookHD assigns a codebook to each quantized level, where each codebook represents using $log_2 q$ bits (Figure 5). For example, for an application with $q = 4$ quantization levels $\{f_1^q, f_2^q, f_3^q, f_4^q\}$, the quantization levels are assigned to $\{00, 01, 10, 11\}$ codebooks. This simplifies the costly lookup search with simple memory access. The concatenation of the codebooks in a chunk is a direct address to a memory row that pre-stored the encoded hypervector. This significantly reduces the cost of encoding in feature chunks. To encode a complete feature vector, LookHD first accesses all chunk hypervectors in parallel; then, it combines them using a set of randomly generated position, **P**, hypervectors (explained in Equation 3).

D. LookHD Training

Although LookHD accelerates the encoding, the main advantage of LookHD appears in training. In conventional HDC, training is implemented by sequentially adding the encoded hypervectors. In contrast, LookHD pre-stores all possible encoded hypervectors and counts the number of times that each one occurs during the training. This simplifies the training process to the multiplication of counter values with pre-stored encoded hypervectors once at the end of the training.

Figure 6 shows the general structure of LookHD performing the training. The training starts with encoding feature vectors to high dimensional space. LookHD implement encoding by quantizing the feature values (*A*). During quantization, each feature value is assigned to the closest quantized levels. This implements by subtracting the feature value from all quantized levels and finding a level with the absolute minimum distance. Depending on the selected level, each feature value is assigned to one of the codebooks (*B*). The concatenation of the codebooks in a chunk is a direct address to pre-stored encoded hypervector (*C*). Instead of looking up the encoded hypervector, LookHD assigns a counter to each chunk. The

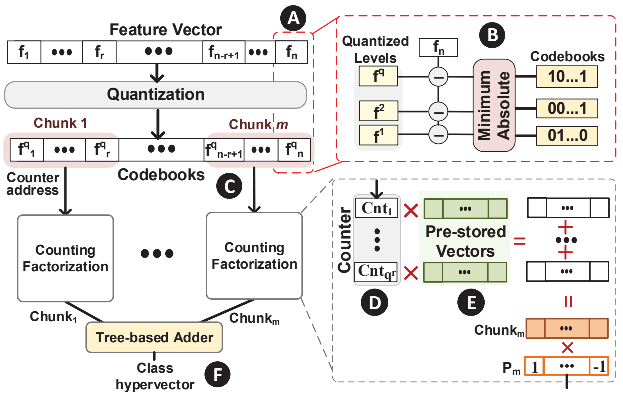


Fig. 6. LookHD training using lookup encoding.

size of the counter is q^r, which is the same as the number of pre-stored chunk hypervectors (D). Thus, each counter row can be accessed using the same codebooks used to access encoded hypervectors. For all data points in a class, LookHD increments the counter values in all chunks without performing the encoding.

After covering all data points in a class, LookHD multiplies the counter values to pre-stored chunk hypervectors. For each chunk, the multiplication results are accumulated to generate the *chunk hypervector* (E). Next, each chunk hypervector multiplies with its corresponding **P** hypervector, and the results are accumulated over all chunks in order to generate a class hypervector (F). The same procedure repeats for all data points in the training dataset in order to generate a hypervector for each existing class. This approach significantly accelerates the HDC training, since LookHD does not need to pay the encoding cost for each data point. In addition, factorizing the values using the counting approach significantly reduces the number of required additions.

IV. LookHD Inference

During the training, HDC creates a single hypervector representing each class. These hypervectors store as a trained HDC model and can be used for the rest of the classification task at inference. The main computations of the inference are the encoding and the associative search. In the inference, HDC uses the same encoding module to map a test data point to a hypervector, called *query hypervector*, $\mathbf{H} \in \mathbb{N}^D$. Then, it computes the similarity of the query hypervector with all k class hypervectors, $\{\mathbf{C}_k, \cdots, \mathbf{C}_2, \mathbf{C}_1\}$, where $\mathbf{C}_i \in \mathbb{N}^D$. Using cosine as a similarity metric, we measure the similarity of a query and i^{th} class hypervector using: $\delta\langle \mathbf{H},\ \mathbf{C}_i\rangle = \mathbf{H}.\mathbf{C}_i/|\mathbf{H}||\mathbf{C}_i|$, where δ denotes the cosine similarity. Finally, each query classifies to a class with the highest cosine similarity.

A. Similarity Metric & Model Scalability

As a light-weight classifier, HDC operations need to be hardware friendly, meaning that the HDC model should fit on the on-chip memory of the embedded devices. The similarity computation can perform efficiently using limited available resources. However, HDC uses the cosine similarity for similarity check. Cosine calculates the inner product of a query and a class hypervector divided by class and query magnitude. In practice, this involves calculating three dot products: $\mathbf{H}.\mathbf{C}_i$, $\mathbf{H}.\mathbf{H}$ and $\mathbf{C}_i.\mathbf{C}_i$. To reduce the cost of cosine computation, we pre-normalize the class hypervectors to their magnitudes, $\mathbf{C'}_i = \mathbf{C}_i/|\mathbf{C}_i|$, once after the training. This enables HDC to ignore repeatedly calculating the class magnitudes for every query. The goal of HDC inference is to find a class hypervector with the highest similarity to a query, i.e., not measuring the absolute cosine values. Thus, we can ignore calculating the query magnitude, $|\mathbf{H}|$, since it is common among all classes. The above two facts simplify the cosine similarity to calculating a dot product between the hypervectors: $\delta\langle \mathbf{H}, \mathbf{C}_i\rangle = \mathbf{H}.\mathbf{C}'_i$.

Although dot product reduces the cost of cosine similarity by $1/3$, the similarity check is still expensive due to many multiplications involved in the dot product. For example, for speech recognition with 26 classes and $D = 10{,}000$, a single query search involves in $26 \times 10{,}000$ multiplications. The model size scalability is another issue in HDC. HDC stores a single hypervector representing each class. This results in increasing the model size by the number of classes. For example, speech recognition with $k = 26$ classes has $13\times$ larger model size as compared to face recognition, which has only $k = 2$ classes.

B. Model Compression

Here, we propose a novel approach to compress the HDC model and address the model scalability issue. LookHD exploits mathematical orthogonality of random hypervectors in order to compress the HDC model and reduce the computation cost. Instead of using k hypervectors to represent the trained model of an application with k classes, LookHD combines all class hypervectors and represents them using a single one. The combined hypervector needs to store the information of all class hypervectors. Similar to the approach used for lookup-based encoding (explained in Section III-D), LookHD combines the class hypervectors while preserving the information of each individual class. LookHD generates k random hypervectors, $\{\mathbf{P'}_1, \mathbf{P'}_2, \cdots, \mathbf{P'}_k\}$, where $\mathbf{P'_i} \in \{-1, 1\}^D$. Since these hypervectors are generated randomly, they will have nearly orthogonal distribution. Using these hypervectors, we can uniquely store the information of each existing class in a combined hypervector as:

$$\mathbf{C} = \mathbf{P'}_1 * \mathbf{C}_1 + \mathbf{P'}_2 * \mathbf{C}_2 + \ldots + \mathbf{P'}_k * \mathbf{C}_k \qquad (4)$$

Figure 7 shows how LookHD creates a combined class hypervector using the trained class hypervectors and a set of $\mathbf{P'}$ hypervectors. Regardless of the number of classes, this approach reduces the HDC model size to a single hypervector. In the inference, HDC checks the similarity of a query with a combined class hypervector by calculating dot product between them:

$$\mathbf{H}.\mathbf{C} = \mathbf{H}.(\mathbf{P'}_1 * \mathbf{C}_1 + \mathbf{P'}_2 * \mathbf{C}_2 + \ldots + \mathbf{P'}_k * \mathbf{C}_k)$$

Next, we can find the similarity of a query and i^{th} class hypervector using:

$$argmax_{j=1:k}\{\delta\langle \mathbf{P'}_i * (\mathbf{H}.\mathbf{C}), \mathbf{C}_j\rangle\}$$

In LookHD, calculating the dot product of $\mathbf{P'}_i$ with $\mathbf{H}.\mathbf{C}$ does not involve multiplication. In hardware, it can be implemented

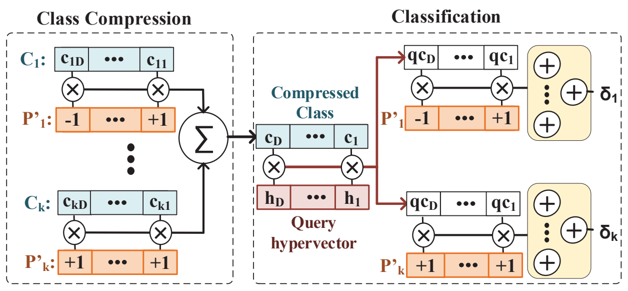


Fig. 7. LookHD model compression and classification.

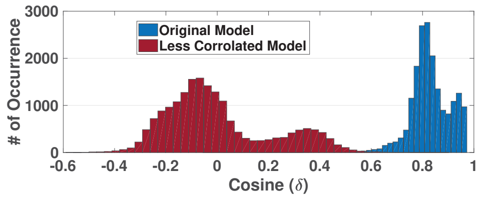


Fig. 8. Cosine distribution on original and less-correlated HDC model.

by changing the sign of $\mathbf{H}.\mathbf{C}$ elements on the dimension that $\mathbf{P}'_i$ has -1 values. This significantly accelerates the LookHD inference by reducing the number of multiplications. We explain the details of hardware implementation in Section V-B.

C. Compression Noise

Depending on the number of classes, the model compression may affect LookHD classification accuracy. Assume we calculate the similarity of a query with i^{th} class:

$$(\mathbf{H}.\mathbf{C})\cdot\mathbf{P}'_i = \underbrace{\mathbf{H}.\mathbf{C}_i * (\mathbf{P}'_i\cdot\mathbf{P}'_i)}_{Signal} + \underbrace{\sum_{j,\forall j\neq i} \mathbf{H}.\mathbf{C}_j * (\mathbf{P}'_i\cdot\mathbf{P}'_j)}_{Noise}. \quad (5)$$

where $\mathbf{P}'_i\cdot\mathbf{P}'_i$ is equal to D, since each element of the base hypervector is either 1 or -1, while $\mathbf{P}'_i\cdot\mathbf{P}'_j$ is almost zero. To quantify the quality of model compression, we defined the signal to noise ratio as a quality metric. This metric determines the noise in the cosine similarity of each class. Although the noise is minimal, in HDC, the class hypervectors are highly correlated, and even small noise may change their ranking during the maximum similarity check. The blue bars in Figure 8 show the distribution of the cosine similarity on ACTIVITY [40] application. The results are reported over 1000 test data. Our result shows that class hypervectors are highly correlated since all cosine similarities are distributed in a range of 0.9 to 1. This increases the sensitivity of the model as a small noise can change the ranking of the top class during the similarity check.

Here, we develop a method that reduces the correlation between the class hypervectors by removing the common information from the classes. LookHD gets the average of the trained class hypervectors ($\mathbf{C}_{ave} = 1/k\sum_{j\in k}\mathbf{C}_i$), and then modifies each class hypervector using:

$$\mathbf{C}'_i = \mathbf{C}_i - \mathbf{C}_{ave}\cdot\delta\langle\mathbf{C}_i,\ \mathbf{C}_{ave}\rangle, \quad \forall i\in k$$

The red bars in Figure 8 show the cosine similarity distribution of the modified class hypervectors. The results indicate that the new model has much wider cosine distribution. This significantly reduces the impact of noise coming from model compression on the similarity measurement. In Section VI-G, we explore the impact of model compression on HDC efficiency-accuracy.

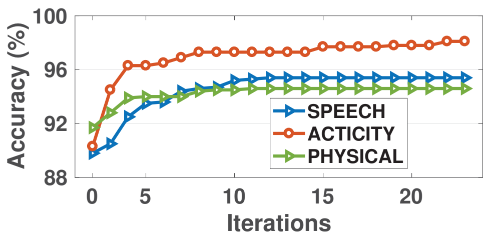


Fig. 9. LookHD classification accuracy during different retraining iterations.

D. Retraining

Retraining happens after the initial HDC training. Retraining has a very similar procedure as the inference. As we explained in Section II-B, the retraining needs to iteratively check the similarity of the encoded train data with the HDC model and accordingly updates the model. The similarity search takes a major portion of the retraining cost. LookHD accelerates the retraining by performing the similarity search over the compressed model. For each mispredicted data, LookHD looks at $\mathbf{P}'$ hypervectors an incorrectly predicted class ($\mathbf{P}'_{wrong}$) and a class that the query belongs to it ($\mathbf{P}'_{correct}$). LookHD accordingly updates the compressed class hypervectors using:

$$\widetilde{\mathbf{C}} = \mathbf{C} + \mathbf{P}'_{correct} * \mathbf{H} - \mathbf{P}'_{wrong} * \mathbf{H}$$

where $\mathbf{P}' \in \{\mathbf{P}'_1,\ldots,\mathbf{P}'_k\}$. LookHD performs the retraining on the HDC model for a few iterations. Figure 9 shows the LookHD classification accuracy during different retraining iterations for three applications. The number of iterations depends on the application, but it is usually about ten iterations to ensure high enough accuracy.

V. Hardware Acceleration

A. Training Acceleration

Figure 10 shows the hardware implementation of LookHD training consisting of four main steps:

Quantization: As we explained in Section III-D, LookHD combines the training with the encoding module. Our approach reads a feature vector and quantizes it by subtracting each feature value from the quantized levels (Figure 10a). Each feature is assigned to a quantized level, which it has the closest absolute distance with it. Finally, each feature gets one of the q codebooks depending on the feature value. This computation is performed in parallel for all features using FPGA Lookup Tables (LUTs) and Flip-Flops (FFs) resources.

Parallel Counting: For each chunk, we use a register array with q^r length and a single counter array to keep track of the number of times that each pre-stored hypervector repeats during the training (Figure 10a). The concatenation of the codebooks in a chunk is a direct address to a counter that increments the corresponding register. Our implementation reads the selected register, increments it using the chunk counter, and writes the result back to the same address. This process can perform in parallel for all chunks.

Weighted Accumulation: After covering all data points in a class, our implementation multiplies the counter values with pre-stored encoded hypervectors stored in BRAM blocks.

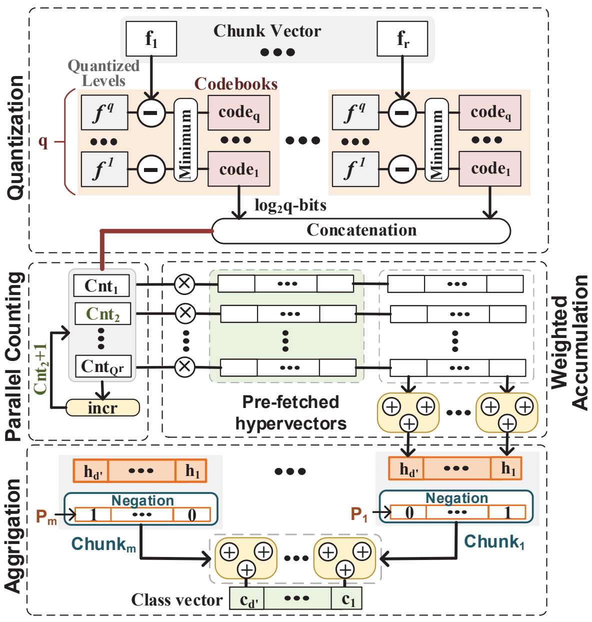


Fig. 10. Hardware acceleration of LookHD training.

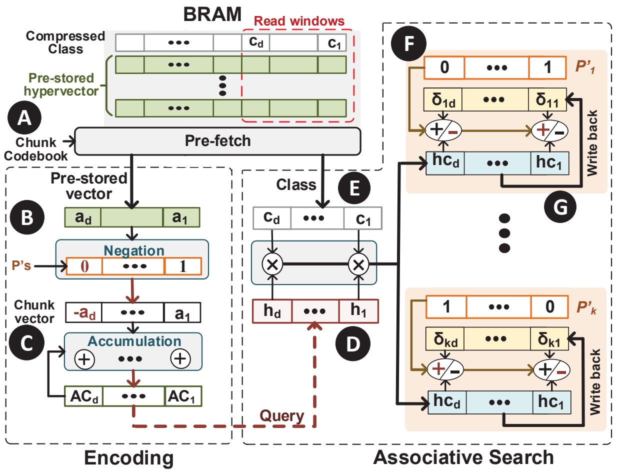


Fig. 11. Hardware acceleration of LookHD inference.

When the size of pre-stored hypervectors is larger than BRAM capacity, LookHD stores the hypervectors in a RAM block. In that case, the performance of computation is limited by the RAM bandwidth. In this paper, we select the chunk size and the number of quantized levels small enough to ensure that the pre-stored hypervectors can fit in BRAM. To provide maximum data locality, our implementation is accessed to the first d elements of all q^r pre-stored hypervectors in BRAM, and multiplies them with the counter values in all chunks. This multiplication can happen using LUTs and FFs, since each element of pre-stored hypervector has only $log_2 r$ bits. Finally, the results of multiplications in all d dimensions are accumulated in a tree-based adder. To maximize the number of dimensions, we use both Digital Signal Processing (DSPs) and available LUTs/FFs resources for this accumulation. The number of d dimensions that can be processed in parallel depends on the number of chunks, m, and counter size, q^r.

Chunk Aggregation: Finally, the generated hypervectors in all chunks need to be combined together using, **P**, position hypervectors (Figure 10). Since the position hypervectors are bipolar, they need more than a single bit for data representation. Our approach represents the position hypervectors with binary values, where -1 elements are replaced with 0. Instead of multiplying the position and chunk hypervectors, we use position hypervector to change the sign of the chunk hypervectors. All elements of a position hypervector with ”0” value flip the sign of the chunk element, while elements with ”1” value keep the sign of the corresponding chunk element the same. Finally, the chunk hypervectors that passed through the negation block are accumulated using a tree-based adder. We reuse the same hardware to generate each class.

B. Inference Acceleration

Figure 11 shows the FPGA implementation of the LookHD inference consisting of the encoding and associative search modules. In order to get maximum throughput, LookHD runs encoding and associative search in a pipeline. When the encoding module creates the d' dimensions of a query hypervector, the associative search is performed on the previous d' dimensions generated by the encoding module. Since the encoding and associative search are majorly using different FPGA resources, this pipeline structure maximizes the throughput and resource utilization. Here, we explain how to implement encoding and associative search in FPGA.

Encoding Acceleration: Similar to the training, the first step of encoding is to quantize each feature value to one of the possible levels and represent it using $Log_2 q$ bits codebook (A). LookHD accesses to the encoded hypervectors in all chunks in parallel (B). Each chunk hypervector needs to be multiplied by the corresponding **P** hypervector. Similar to the training, we store **P** hypervectors as binary representation and use them as input to the negation block (B). All **P** hypervector dimensions with ”0” value flip the sign of the corresponding encoded hypervector. Finally, all d dimensions of vectors are accumulated together. Since each dimension of encoded hypervectors only has $Log_2 q$ bits, all the computation can perform using LUT/FF blocks (C).

Associative Search Acceleration: The computation of the associative search summarizes into three main steps (Figure 11): (i) multiplying the query and class hypervectors, (ii) multiplying the product result with different position hypervectors corresponding to each class, and (iii) accumulating all elements for each class. Our implementation exploits DSPs to multiply query and class hypervectors (D). However, since the number of available DSPs are much lower than the hypervector dimensions, this similarity happens serially over d dimensional windows. The similarity check, i.e., dot product, starts sequentially by multiplying d dimensions of a query and compressed class hypervector. Our implementation represents $\mathbf{P}'$ hypervectors using binary values, where the -1 elements in bipolar vector represent 0 element (Ⓔ). To parallelize the associative search for all classes, our implementation reads the first d dimensions of the position hypervector, $\mathbf{P}'$, and assigns them as control signals to DSP blocks. The $\mathbf{P}'$ dimensions with 0 values configure DSPs to subtraction, while $\mathbf{P}'$ elements with 1 elements configure DSP to perform an addition operation (Ⓕ). The accumulation of the product elements happens in parallel for all classes. Finally, the results of accumulation are written back to the same product vector (Ⓖ). This operation repeats iteratively by moving the read windows over the query and class hypervectors until it covers all D dimensions. Finally, the results of all d dimensions are accumulated to calculate the result of the dot product. The size of windows d depends on the number of classes and the number of DSPs available on FPGA. For example, for activity recognition application with $k = 6$ classes, our implementation can parallelize the computation on $d = 64$ dimensions.

C. Retraining Acceleration

The retraining can be accelerated using the similar hardware used for associative search. First, we check the similarity of a query and a compressed trained model using the hardware shown in Figure 11. When a query is misclassified, retraining updates the HDC model by adding and subtracting the query from two classes. This update happens once after going over the entire training dataset. Our implementation applies all modifications on a copy of the compressed model while using the original model for inference tasks. Instead of applying one addition and subtraction to update the model, our implementation first calculates the $\Delta\mathbf{P}' \cdot \mathbf{H}$ term and adds it to the compressed model. The subtraction of two bipolar $\Delta\mathbf{P}' = \mathbf{P}'_{correct} - \mathbf{P}'_{wrong}$ can get -2, +2, 0 values. Since in hardware we represent the $\mathbf{P}'$s using binary values, we can decide to update each query elements depending on the $\mathbf{P}'$ bits. The following equating shows how $\mathbf{H} * \delta\mathbf{P}'$ can be modeled using negation and shift operations:

$$\mathbf{C} = \mathbf{C} + \Delta\mathbf{P}' \cdot \mathbf{H}$$

$$\Delta\mathbf{P}' \cdot \mathbf{H} = \begin{cases} -h >>, & \text{if } (\mathbf{P}'_{correct}, \mathbf{P}'_{wrong}) = (0,0) \\ h, & \text{if } (\mathbf{P}'_{correct}, \mathbf{P}'_{wrong}) = (0,1) \ or \ (1,0) \\ h >>, & \text{if } (\mathbf{P}'_{correct}, \mathbf{P}'_{wrong}) = (1,1) \end{cases}$$

where h is the element of the query hypervector and $>>$ defines as a single right shift. Depending on the $\mathbf{P}'$s elements, each query element stays the same, shifted, or shifted and get a filliped sign bit. This functionality can apply to a query using LUT/FF blocks. After that, the result of the hypervector will be added to a created copy of the compressed model. To maximize the throughput, the similarity check and model update modules are implemented in a pipeline structure. For all tested applications, the number of DSPs limits retraining throughput.

VI. Evaluation

A. Experimental Setup

We implement LookHD training and testing on two platforms: FPGA and CPU. For FPGA, we describe the LookHD functionality using Verilog and synthesize it using Xilinx Vivado Design Suite [46]. The synthesis code has been implemented on the Kintex-7 FPGA KC705 Evaluation Kit using 5ns clock frequency. For CPU, the LookHD code has been written in C++ and optimized for performance. The code has been implemented on the ARM Cortex A53 CPU, and its power is measured using Hioki 3334 power meter.

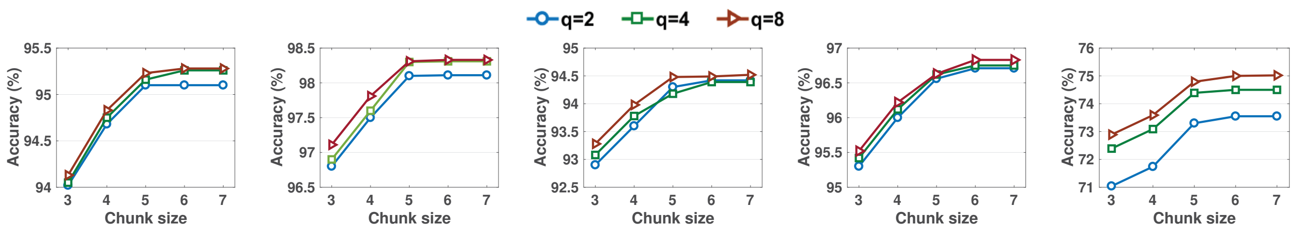
We compare the accuracy and efficiency of LookHD with baseline HDC [37], [38] implemented on both CPU and FPGA. To have a fair comparison and show the true benefits of our lookup based approach, we considered an optimized implementation of the HDC algorithm [38] explained in Section II as the baseline. The training is implemented by encoding the data points to high-dimensional space and adding the encoded hypervectors in a pipelined stage. In the inference, the baseline runs on the same hardware as LookHD (Section V-B), but uses the original encoding and performs similarity check over the non-compressed model. We evaluate LookHD on benchmarks range from relatively small datasets collected in a small IoT network to a large dataset that includes hundreds of thousands of face images. *SPEECH:* voice recognition [39], *ACTIVITY:* activity recognition using mobile device [40], *PHYSICAL:* physical monitoring using IMU sensors [41], *FACE:* face recognition [42], and *EXTRA:* phone position recognition [43].

B. LookHD Accuracy

LookHD trains the model by splitting the feature vector and quantizing the level hypervectors to discrete levels. Figure 12 shows the impact of chunk size and the number of quantized levels on the LookHD classification accuracy ($D = 2000$). We compare LookHD accuracy with the baseline HDC algorithm using linear quantization level [37], [47]. The baseline accuracy of each application is listed in the sub-figure title of Figure 12. In LookHD, increasing the chunk size generally improves the classification accuracy. A small chunk degrades the quality of encoding by increasing the number of required $\mathbf{P}$ hypervectors to aggregate the result of different chunks. The best chunk size depends on the distribution of the feature values. Our evaluation shows that for most applications using $r = 5$ is enough to provide acceptable accuracy.

LookHD classification accuracy also depends on the levels of quantization. The larger the number of quantization levels, results in a higher LookHD accuracy. However, with the proposed equalized quantization, the change in accuracy is minor. Our results show that for most applications using $q = 2$ or 4 is enough to ensure acceptable classification accuracy. In contrast, the linear quantization of the existing HDC algorithms [33], [37], [47] results in generating several levels which are not equally used during the classification. This degrades the accuracy by making the classification task more complicated. Our evaluation shows that LookHD with $q = 2$ and $q = 4$ achieve, on average, 2.1% and 2.4% higher accuracy than the baseline HDC using a non-binarized model.

Table II illustrates the impact of the hypervector dimensions on the LookHD classification accuracy using a chunk size of $r = 5$ for the quantization levels listed in the table. The results indicate the robustness of LookHD to the reduction of hypervector dimensions. For example, LookHD with $D = 2000$



(a) SPEECH(**HD=91.4%**) (b) ACTIVITY(**HD=94.6%**) (c) PHYSIC(**HD=91.3%**) (d) FACE(**HD=94.1%**) (e) EXTRA(**HD=70.6%**)

Fig. 12. Impact of the chunk size and the quantization levels on the LookHD classification accuracy.

TABLE II
IMPACT OF DIMENSIONALITY ON LOOKHD ACCURACY.

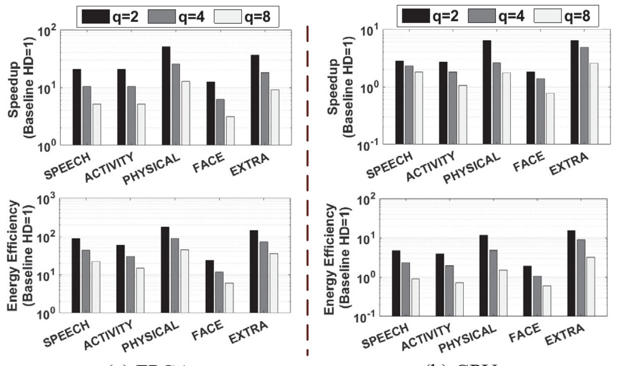
D	q	1000	2000	4000	8000	10,000
SPEECH	4	94.8%	95.2%	95.3%	95.5%	95.5%
ACTIVITY	4	97.3%	97.9%	97.9%	98.0%	98.2%
PHYSICAL	2	91.4%	92.9%	92.9%	93.1%	93.1%
FACE	2	95.7%	96.5%	96.6%	96.7%	96.8%
EXTRA	4	72.5%	73.3%	73.3%	73.4%	73.4%

dimensions provides the similar accuracy as HDC using full $D = 10,000$ dimensions, i.e., less than 0.3% quality loss. In the rest of the paper, we show the energy/performance results of LookHD and the baseline HDC for $D = 2000$ dimensions.

C. Training Acceleration

We optimize the baseline HDC implementation to maximize training throughput. In the baseline HD, the training consists of an encoding module, which involves several bitwise operations. This significantly improves the efficiency of the FPGA training as compared to the CPU implementation. Our evaluation shows that the FPGA implementation of the baseline HDC is, on average, 830.2× faster and 1,509.4× more energy efficient as compared to CPU. Figure 13 shows the training efficiency on both FPGA and CPU for different LookHD configurations, when the feature values are quantized into $q = 2$, 4 and 8 levels ($r = 5$).

The number of quantization levels presents a tradeoff between training efficiency and classification accuracy. The larger quantization levels slightly improve the classification accuracy, but they significantly degrade the training efficiency. In addition, LookHD memory requirement for training exponentially increases with the number of quantization levels. For example, doubling the quantization levels from $q = 2$ to $q = 4$ ($r = 5$), increases the size of memory requirement to pre-store the encoded hypervectors from $2^5 = 32$ to $4^5 = 1024$ rows. This directly affects the training cost as LookHD requires more logic operations to calculate the counter-vector multiplications and aggregate the chunk hypervectors. Our evaluations show that FPGA-based (CPU-based) implementation of LookHD using $q = 2$ and $q = 4$ levels achieves, on average, 28.3× and 97.4× (3.9× and 7.5×) faster and more energy-efficient training as compared to the state-of-the-art HDC. Similarly, using $q = 4$ results on average 14.1× and 48.7× (2.6× and 3.8×) faster and more energy-efficient training. This higher efficiency comes from the capability of LookHD to simplify the encoding operations and combine it with the training module. LookHD learns from highly quantized input data,



(a) FPGA (b) CPU

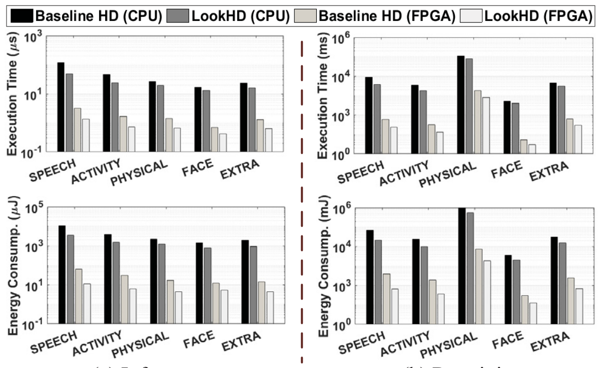
Fig. 13. Speedup and energy efficiency improvement of LookHD training running on different platforms.

which is not possible in the original space. This is because LookHD encoding is non-linear and projects input data with a small difference to relatively isolated data in sparse high-dimensional space. For example, during projection, a single feature difference between two data points could result in mapping them into very different hypervectors in HDC space.

D. Inference Acceleration

In the inference, HDC consists of the encoding and associative search. The encoding maps the input data to high-dimensional space by performing several bitwise operations, while the computation of the associative search is mostly dot product of non-binarized hypervectors. Here, we compare the efficiency of the LookHD and the baseline HDC algorithm [33], [37], [47] on CPU and FPGA platforms.

LookHD enhances the encoding efficiency by pre-storing all possible chunk hypervectors in a memory. This simplifies the encoding module by aggregating the pre-stored chunk hypervectors using m number of position (**P**) hypervectors. Since $m << n$, this aggregation can happen much faster than original encoding, where m and n are the number of chunks and the number of features, respectively. In the associative search, combining the class hypervectors reduces the model size and the number of required multiplications. The small model size further improves the computation efficiency by (i) significantly reducing the number of multiplications, (ii) localizing the computation to existing cores. As Figure 14a shows, this advantage is more evident on FPGA, since it has smaller memory and computing resources to parallelize the computation. On average, on CPU, LookHD achieves, on



(a) Inference (b) Retraining

Fig. 14. Execution time and energy consumption of LookHD and the baseline HDC running (a) a single query in the inference (b) a single retraining iteration.

average, 1.7× faster and 2.3× more efficient computation than the baseline HDC.

On FPGA, the LookHD encoding and associative search blocks are working in the pipeline. As we explained in Section V-B, when associative search checks the similarity of the d dimensions of a query and class hypervectors, the encoding module generates the next d dimensions. The encoding module is implemented using LUTs and FFs, while the associative search mostly utilizes DSPs. Since these two modules use different resources, the value of d depends on the resource constraints in each of the modules. Since the number of FPGA (Kintex-7) DSPs are limited to 840, in all tested cases, the associative search limits the d' values. For example, for ACTIVITY and FACE with 12 and 2 classes, the associative search can process at most $d' = 64$ and $d' = 256$ dimensions in parallel, respectively. The results show that FPGA implementation of LookHD is, on average, 2.2× faster and 4.1× more energy efficient as compared to the baseline HD. As compared to CPU-based implementation of LookHD, the FPGA-based implementation is, on average, 122.9× faster and 238.6× more energy efficient.

E. Retraining

Figure 14b compares the execution time and energy consumption of LookHD with the baseline HDC during a single iteration of the retraining. Similar to inference, the retraining uses the associative search to check the similarity of each training data with the trained model. LookHD updates a copy of the compressed model for each misclassified data. In our evaluations, for each application, we consider the average number of updates during the entire training iterations. The results show that the efficiency of retraining depends on the number of classes and the number of required updates. For SPEECH with the largest number of classes, i.e., 26 classes, LookHD provides the maximum advantage. Our results show that LookHD retraining is, on average, 2.4× and 4.5× (1.8× and 2.3×) faster and more energy-efficient than the baseline HDC running on FPGA (CPU), respectively.

F. LookHD vs. GPU Implementation

All proposed algorithm-hardware operations are general and can be implemented on any platform with bit-level granularity, including low-power or high-performance FPGA/ASIC. However, we target a low-power platform as HDC provides several features that make it promising for real-time learning on embedded devices. For example, HDC supports single-pass or few-pass training on devices with low on-chip memory and computational resources. Although our goal is to run LookHD on embedded platforms (e.g., FPGA or ARM CPU), here we compare it with the powerful GPU architecture to better show the efficiency of LookHD. Table III compares the speedup and energy efficiency of LookHD with the GPU implementation running on NVIDIA GTX 1080 GPU. We have an optimized implementation of HDC using Tensorflow [48]. Unlike the low-power platforms that we aim to eliminate the computation, GPU can provide significant acceleration by enabling parallelism over different dimensions. All results are relative to the energy consumption and execution time of the CPU implementation. Our evaluation shows that although GPU training and inference are 1.5× (1.3×) faster than the FPGA implementation of the baseline HD, LookHD provides 1.1× and 1.5× faster training and inference than GPU. In terms of energy efficiency, LookHD is, on average, 67.5× and 112.7× more energy-efficient than GPU during training and inference, respectively. Table III also shows that LookHD can further improve computation efficiency by reducing hypervector dimensionality. For example, LookHD losing less than 2% quality loss provides 1.21× and 1.25× faster training and inference than LookHD using $D = 2000$ dimensions.

TABLE III
AVERAGE LOOKHD SPEEDUP AND ENERGY EFFICIENCY OVER NVIDIA GPU (NORMALIZED TO CPU)

Platforms		GPU	FPGA	LookHD @0%	LookHD @1%	LookHD @2%
Train	Speedup	121.7×	79.2×	135.9×	158.1×	169.7×
	Energy Efficiency	5.9×	131.0×	397.2×	431.5×	478.3×
Test	Speedup	133.8×	112.7×	197.5×	221.9×	239.9×
	Energy Efficiency	4.3×	148.1×	490.1×	524.4×	563.6×

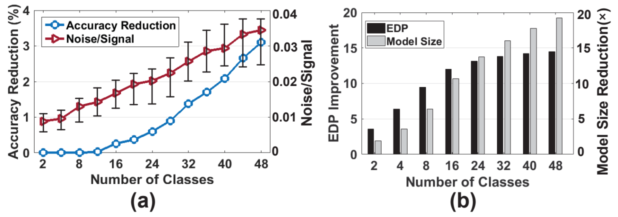


Fig. 15. Impact of LookHD model compression on the accuracy and the similarity noise/signal ratio. (b) LookHD scalability with the number of classes.

G. LookHD Inference Scalability

As a light-weight classifier, HDC is desired to provide a small and scalable model that can be stored and processed on embedded devices with limited resources. In the conventional HDC model, each class is represented using a single hypervector. Therefore, the HDC model size increases linearly with the number of classes. LookHD addresses the model size scalability issue by combining and compressing all class patterns into a single hypervector. Figure 15 shows the tradeoff between the LookHD classification accuracy and computation efficiency when the number of classes increases from k=2 to 48.

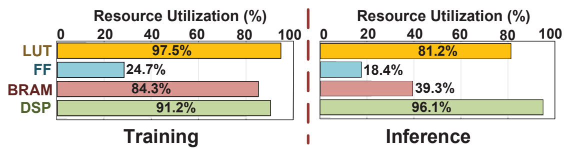


Fig. 16. LookHD resource utilization.

The results are reported for running 1000 queries on randomly generated class hypervectors with Gaussian distribution, where the classes have a similar correlation as five tested models.

Figure 15a shows the impact of the number of classes on the classification accuracy and the average noise to signal ratio of the compressed model. LookHD with more number of classes has higher noise to signal ratio due to the error from non-orthogonality between the $\mathbf{P}'$ hypervectors (shown in Equation 5). This noise can change the rank of the classes during the search for maximum cosine similarity and result in misclassification. Our result shows that LookHD does not lose any accuracy for applications with 12 or fewer classes. However, since the noise in classification statistically grows with the number of classes, LookHD is likely to lose accuracy for applications with more than 12 classes. For example, an application with 26 classes can provide less than 0.8% quality loss as compared to HDC with the non-compressed model. For applications with more than 12 classes, LookHD compresses the trained model into multiple hypervectors in order to eliminate the quality loss. To this end, each compressed hypervector needs to keep the information of less than 12 classes. For example, for an application with 36 classes, LookHD can compress the model into three hypervectors, where each is a combination of 12 hypervectors. This ensures no quality loss while still providing 8.7× smaller model size.

Figure 15b reports the energy-delay product (EDP) improvement of LookHD using the compressed model as compared to the baseline HDC running on the FPGA. Figure 15b also shows the LookHD model size reduction when LookHD compresses the HDC model into a single hypervector. Our result shows that LookHD can achieve 6.9× and 12.0× EDP improvement and smaller model size while providing the same accuracy as the non-compressed model. For applications with more classes, the EDP improvement, and model size improve significantly with minimal impact on the accuracy. For example, for an application with 48 classes, LookHD can achieve 14.6× and 19.2× EDP improvement and smaller model size with about 2% quality loss while compressing the model into a single hypervector. In an exact mode, LookHD compresses the model into four hypervectors, which still results in 10.8× EDP improvement and 8.7× model size reduction, while ensuring the same accuracy as the non-compressed model.

H. Resource Utilization

Figure 16 shows the resource utilization of LookHD FPGA implementation during training and inference phases. The associative memory and encoding module utilize different resources depending on the number of classes and the feature size. Here, we show the resource utilization for SPEECH with $k = 26$ classes and $n = 617$ features. The resources utilized by the encoding module mostly consists of LUTs and FFs, which are used to count and store the pre-fetched base hypervectors. In inference and retraining, the associative memory mostly utilizes DSPs. However, since LookHD stores the compressed model, the associative search also uses LUTs and FFs in order to compute the similarity values. In addition, the model update in LookHD further increases the BRAM and LUT utilization in the training phase. As results in Figure 16 show, for SPEECH inference, the FPGA performance is limited by the number of DSPs, while in training, the LUT utilization is the performance bottleneck. Note that depending on the number of features or classes, applications may have different resource utilization. For example, for FACE application with $k = 2$ classes and $n = 608$ features ($k << n$), the LUTs are the computation bottleneck in both training and inference phases.

TABLE IV
LookHD efficiency vs. MLP on FPGA

		SPEECH	UCIHAR	PAMAP2	FACE	EXTRA
Train	**Speedup**	16.6×	19.1×	26.8×	31.7×	24.7×
	Energy Efficiency	30.4×	41.2×	48.5×	61.3×	42.8×
Test	**Speedup**	7.9×	10.8×	12.6×	17.3×	11.9×
	Energy Efficiency	3.7×	4.9×	5.4×	6.3×	6.0×

I. LookHD vs Other Classifiers

Table IV also compares the training/inference efficiency of LookHD with MLP on FPGA. We used DNNWeaver V2.0 [49] for efficient implementation of the NN inference, and FPDeep [50] for NN training on a single FPGA device. FPGA implementations are optimized to maximize performance by utilizing FPGA resources. All results listed in Table IV are relative to MLP performance and energy efficiency. Note that we compare the baseline MLP and LookHD, since several existing optimizations, e.g., model binarization or pruning [51], [52], can be applied to both methods. During training, LookHD achieves, on average, 23.1× faster and 43.6× more energy-efficient computation as compared to FPGA-based MLP implementation, respectively. The high efficiency of LookHD in training comes from: (i) LookHD capability in creating an initial model that significantly lowers the number of required retraining iterations. (ii) It eliminates the costly gradient descent for the model update. This results in a higher LookHD efficiency, even in terms of a single training iteration. In inference, LookHD provides 11.7× faster and 5.1× higher energy efficiency as compared to FPGA-based MLP implementation. This higher inference efficiency comes from LookHD ability in reducing the number of required resources (multiply-add), on average, by 38.1× compared to the equivalent MLP. LookHD model compression further reduces memory footprints and provides an average 63.2× smaller model size compared to the MLP.

VII. Related Work

Since the neuroscientist P. Kanerva introduced the field of hyperdimensional computing [10], prior research have applied the idea into diverse cognitive tasks such as robotics [14], [53], analogy-based reasoning [54], latent semantic analysis [44], text classification [55], genome pattern matching [15], activity recognition [56], prediction from multimodal sensor fusion [17], [57], speech recognition [58]. For example, work in [14] showed how to use HDC mathematics to enables robots to model the human-like memory. The work in [18], [59] showed an HD application for feature vector classification. These approaches assume that HDC learning tasks are performed in the binary domain. However, working with binary vectors provides significantly lower accuracy when running on practical workloads (on average, 17.5% than LookHD). In contrast, LookHD works with the non-binary model and provides significantly higher accuracy on complex classification problems.

Several existing works have presented the hardware for efficiently processing HDC, including both FPGA and in-memory accelerators [21], [22], [23], [37], [60], [61], [62]. Work in [63] implemented HDC on FPGA, but it only works with binary hypervectors with applications limited to the classification of text and time-series signals. Work in [37], [64] designed a new FPGA-based architecture to accelerate the classification task. However, it requires an explicit encoding and training process, resulting in low computation efficiency. Work in [21] showed three digital, resistive, and analog circuits to accelerate the computation of Hamming distance similarity search in the HDC inference. Although these approaches accelerate the HDC computation, the encoding still takes a dominant part of the training procedure. Work in [22] designed and fabricated an HDC chip accelerator using emerging memory devices. However, this architecture only supports the binary model, and its application is limited to the text classification. To the best of our knowledge, LookHD is the first efficient architecture that systematically eliminates the cost of encoding from HDC, with no need for hardware acceleration. In addition, LookHD addresses the model/inference scalability of all prior HDC work by compressing the model and performing the training and inference on the compressed model.

VIII. Conclusion

In this paper, we propose LookHD to address two major issues of the HDC-based systems: the costly encoding and the scalability of the HDC model. LookHD exploits the computation reuse by bounding the number of possible values that the encoded hypervector can take. This eliminates the encoding computation and simplifies the encoding to a simple memory lookup. In the inference, LookHD uses the mathematical orthogonality of random vectors to significantly compress the HDC model and reduce the cost of inference. We accordingly design an embedded architecture to accelerate LookHD on FPGA. Our evaluations show that LookHD achieves 28.3× speedup and 97.4× energy efficiency training as compared to the state-of-the-art design.

Acknowledgment

This work was partially supported by Semiconductor Research Corporation (SRC) Task No. 2988.001. Mohsen Imani and Yeseong Kim are co-corresponding authors of the paper.

References

[1] Y. Xiang, A. Alahi, and S. Savarese, “Learning to track: Online multi-object tracking by decision making,” in *2015 IEEE international conference on computer vision (ICCV)*, no. EPFL-CONF-230283, pp. 4705–4713, IEEE, 2015.

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, *et al.*, “Deep speech 2: End-to-end speech recognition in english and mandarin,” in *International Conference on Machine Learning*, pp. 173–182, 2016.

[3] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, *et al.*, “Deep speech: Scaling up end-to-end speech recognition,” *arXiv preprint arXiv:1412.5567*, 2014.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[5] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[6] M. Denil, B. Shakibi, L. Dinh, N. De Freitas, *et al.*, “Predicting parameters in deep learning,” in *Advances in neural information processing systems*, pp. 2148–2156, 2013.

[7] A. Zaslavsky, C. Perera, and D. Georgakopoulos, “Sensing as a service and big data,” *arXiv preprint arXiv:1301.0159*, 2013.

[8] Y. Sun, H. Song, A. J. Jara, and R. Bie, “Internet of things and big data analytics for smart and connected communities,” *IEEE Access*, vol. 4, pp. 766–773, 2016.

[9] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, “Future internet: the internet of things architecture, possible applications and key challenges,” in *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pp. 257–260, IEEE, 2012.

[10] 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.

[11] P. Kanerva, *Sparse distributed memory*. MIT press, 1988.

[12] B. Babadi and H. Sompolinsky, “Sparseness and expansion in sensory representations,” *Neuron*, vol. 83, no. 5, pp. 1213–1226, 2014.

[13] A. Cano, N. Matsumoto, E. Ping, and M. Imani, “Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021.

[14] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, “Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception,” *Science Robotics*, vol. 4, no. 30, p. eaaw6736, 2019.

[15] Y. Kim, M. Imani, N. Moshiri, and T. Rosing, “Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 115–120, IEEE, 2020.

[16] B. Khaleghi, M. Imani, and T. Rosing, “Prive-hd: Privacy-preserved hyperdimensional computing,” *arXiv preprint arXiv:2005.06716*, 2020.

[17] O. Räsänen and S. Kakouros, “Modeling dependencies in multiple parallel data streams with hyperdimensional computing,” *IEEE Signal Processing Letters*, vol. 21, no. 7, pp. 899–903, 2014.

[18] M. Imani, J. Messerly, F. Wu, W. Pi, and T. Rosing, “A binary learning framework for hyperdimensional computing,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 126–131, IEEE, 2019.

[19] A. Cano, Y. Kim, and M. Imani, “A framework for efficient and binary clustering in high-dimensional space,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021.

[20] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, “Dual: Acceleration of clustering algorithms using digital-based processing in-memory,” in *53rd International Symposium on Microarchitecture (MICRO)*, pp. 1–14, IEEE/ACM, 2020.

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

[22] T. Wu, P. Huang, A. Rahimi, H. Li, J. Rabaey, P. Wong, and S. Mitra, “Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study,” in *IEEE Intl. Solid-State Circuits Conference (ISSCC)*, IEEE, 2018.

[23] H. Li, T. F. Wu, A. Rahimi, K.-S. Li, M. Rusch, C.-H. Lin, J.-L. Hsu, M. M. Sabry, S. B. Eryilmaz, J. Sohn, *et al.*, “Hyperdimensional

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

[24] "WebFeet Research is Implementing Hyperdimensional Computing on FPGA enhanced with NVDIMM." http://www.webfeetresearch.com/.

[25] M. Lázaro-Gredilla, D. Lin, J. S. Guntupalli, and D. George, "Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs," *arXiv preprint arXiv:1812.02788*, 2018.

[26] "Numenta Use Hyperdimensional Computing as Hierarchical Temporal Memory." https://numenta.org/.

[27] S. Ahmad and J. Hawkins, "Properties of sparse distributed representations and their application to hierarchical temporal memory," *arXiv preprint arXiv:1503.07469*, 2015.

[28] G. Karunaratne, M. L. Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *arXiv preprint arXiv:1906.01548*, 2019.

[29] Y. Wu, G. Wayne, A. Graves, and T. Lillicrap, "The kanerva machine: A generative distributed memory," *arXiv preprint arXiv:1804.01756*, 2018.

[30] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 505–516, 2014.

[31] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2016.

[32] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 120, ACM, 2015.

[33] 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.

[34] P. Kanerva, J. Kristofersson, and A. Holst, "Random indexing of text samples for latent semantic analysis," in *Proceedings of the 22nd annual conference of the cognitive science society*, vol. 1036, Citeseer, 2000.

[35] T. F. Wu, H. Li, P.-C. Huang, A. Rahimi, J. M. Rabaey, H.-S. P. Wong, M. M. Shulaker, and S. Mitra, "Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pp. 492–494, IEEE, 2018.

[36] M. Imani, Y. Kim, S. Riazi, J. Messerly, P. Liu, F. Koushanfar, and T. Rosing, "A framework for collaborative learning in secure high-dimensional space," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 435–446, IEEE, 2019.

[37] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 53–62, ACM, 2019.

[38] A. Rahimi, T. F. Wu, H. Li, J. M. Rabaey, H.-S. P. Wong, M. M. Shulaker, and S. Mitra, "Hyperdimensional computing nanosystem," *arXiv preprint arXiv:1811.09557*, 2018.

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

[40] 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.

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

[42] Y. Kim, M. Imani, and T. Rosing, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*, pp. 25–32, IEEE, 2017.

[43] Y. Vaizman, K. Ellis, and G. Lanckriet, "Recognizing detailed human context in the wild from smartphones and smartwatches," *IEEE Pervasive Computing*, vol. 16, no. 4, pp. 62–74, 2017.

[44] P. Kanerva, J. Kristofersson, and A. Holst, "Random indexing of text samples for latent semantic analysis," in *Proceedings of the 22nd annual conference of the cognitive science society*, vol. 1036, Citeseer, 2000.

[45] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[46] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[47] 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.

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

[49] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[50] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt, "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 81–84, IEEE, 2018.

[51] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[52] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 548–560, 2017.

[53] S. Jockel, "Crossmodal learning and prediction of autobiographical episodic experiences using a sparse distributed memory," 2010.

[54] p. kanerva, "What we mean when we say" what's the dollar of mexico?": Prototypes and mapping in concept space," in *AAAI fall symposium: quantum informatics for cognitive, social, and semantic processes*, vol. 1036, Citeseer, 2010.

[55] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey, "Hyperdimensional computing for text classification," *Design, Automation Test in Europe Conference Exhibition (DATE), University Booth*, 2016.

[56] Y. Kim, M. Imani, and T. S. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, pp. 1–6, 2018.

[57] 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.

[58] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *International Conference on Rebooting Computing (ICRC)*, pp. 1–6, IEEE, 2017.

[59] M. Imani, J. Morris, J. Messerly, H. Shu, Y. Deng, and T. Rosing, "Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[60] S. Salamat, M. Imani, and T. Rosing, "Accelerating hyperdimensional computing on fpgas by exploiting computational reuse," *IEEE Transactions on Computers*, 2020.

[61] S. Gupta, J. Morris, M. Imani, R. Ramkumar, J. Yu, A. Tiwari, B. Aksanli, and T. Š. Rosing, "Thrifty: Training with hyperdimensional computing across flash hierarchy," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2020.

[62] M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, and T. Rosing, "Quanthd: A quantization framework for hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[63] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *arXiv preprint arXiv:1807.08583*, 2018.

[64] M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, and T. Rosing, "Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 190–198, IEEE, 2019.