Accelerating Hyperdimensional Computing on FPGAs by Exploiting Computational Reuse

Sahand Salamat, *Student Member, IEEE*, Mohsen Imani, *Student Member, IEEE*, and Tajana Rosing, *Fellow, IEEE*

Abstract—Brain-inspired hyperdimensional (HD) computing emulates cognition by computing with long-size vectors. HD computing consists of two main modules: encoder and associative search. The encoder module maps inputs into high dimensional vectors, called hypervectors. The associative search finds the closest match between the *trained model* (set of hypervectors) and a *query* hypervector by calculating a similarity metric. To perform the reasoning task for practical classification problems, HD needs to store a non-binary model and uses costly similarity metrics as *cosine*. In this article we propose an FPGA-based acceleration of HD exploiting Computational Reuse (`HD-Core`) which significantly improves the computation efficiency of both encoding and associative search modules. `HD-Core` enables computation reuse in both encoding and associative search modules. We observed that consecutive inputs have high similarity which can be used to reduce the complexity of the encoding step. The previously encoded hypervector is reused to eliminate the redundant operations in encoding the current input. `HD-Core`, additionally eliminates the majority of multiplication operations by clustering the class hypervector values, and sharing the values among all the class hypervectors. Our evaluations on several classification problems show that `HD-Core` can provide $4.4\times$ energy efficiency improvement and $4.8\times$ speedup over the optimized GPU implementation while ensuring the same quality of classification. `HD-Core` provides $2.4\times$ more throughput than the state-of-the-art FPGA implementation; on average, 40 percent of this improvement comes directly from enabling computation reuse in the encoding module and the rest comes from the computation reuse in the associative search module.

Index Terms—Brain-inspired computing, hyperdimensional computing, machine learning, FPGA, energy efficiency

1 Introduction

Machine learning algorithms have shown a promising solution in many tasks, including computer vision, voice recognition, natural language processing, and health care [1], [2], [3], [4]. However, existing machine learning algorithms such as Deep Neural Networks (DNNs) are computationally expensive and require an enormous amount of resources to be executed [5], [6], [7]. Moreover, embedded devices (e.g., wearable devices, smartphones, etc.) are often constrained in terms of available processing resources and power budget [8], [9], [10], [11]. Brain-inspired hyperdimensional (HD) computing is a computational paradigm performing energy-efficient cognitive computation with comparable accuracy to computation-intensive machine learning algorithms [12], [13]. Brain performs cognition tasks based on the *patterns of neural activity* that are not readily associated with numbers [14]. HD computing models such neural activity patterns with vectors in high-dimensional space, called hypervectors.

HD computing builds upon a well-defined set of operations with random hypervectors. In addition, it is robust in the presence of faults due to the holographic distribution of the patterns in high-dimensional space (random patterns with i.i.d distributions). HD computing offers a complete computational paradigm that applies to various learning problems, including: analogy-based reasoning [15], latent semantic analysis [16], language recognition [17], prediction from multimodal sensor fusion [18], speech recognition [19], activity recognition [20], DNA sequencing [21], and clustering [22]. In contrast to existing classification algorithms which require significantly complex and costly computation during training and inference [23], [24], [25], HD provides high parallelism with hardware-friendly training and inference operations that can be processed on light-weight embedded devices [13].

All the main steps of HD computing is illustrated in Fig. 1. The first step of HD computing is representing data with hypervectors. The encoding module generates a hypervector for each input as a set of pre-processed features. Encoding module keeps the information of the input features in a hypervector. To train the HD model, inputs belong to a class are encoded and added together to generate a hypervector representing each class. At the inference phase, the incoming input is encoded to a hypervector, called the *query hypervector*, and then the associative search module checks the similarity between the query and each class hypervector. The class with the highest similarity to the query hypervector is selected as the classification result. During training, encoding dominates the entire energy and execution time since the training is a simple addition operation. In the inference, both associative search and encoding steps are computationally complex. Therefore, accelerating HD requires accelerating both

• *The authors are with the Department of Computer Science and Engineering, University of California San Diego, La Jolla, CA 92093. E-mail: {sasalama, moimani, tajana}@ucsd.edu.*

Manuscript received 15 Dec. 2019; revised 22 Apr. 2020; accepted 2 May 2020. Date of publication 6 May 2020; date of current version 9 July 2020. (Corresponding author: Sahand Salamat.)
Digital Object Identifier no. 10.1109/TC.2020.2992662

0018-9340 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

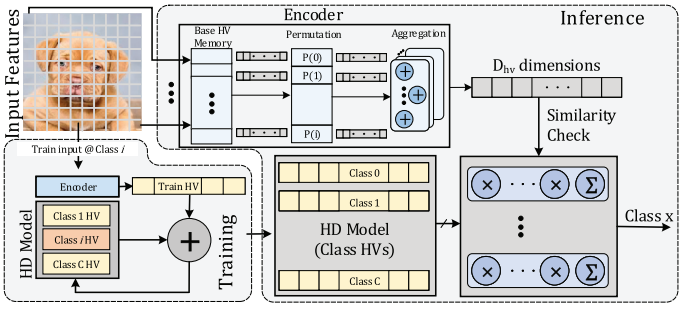

Fig. 1. HD functionality in training and inference phases using encoding and associative search modules.

encoding and associative search using algorithmic and hardware optimizations. In this paper, we propose a algorithmic-hardware co-optimization platform to accelerate both encoding and associative search, resulting in significant acceleration of the HD computing.

To accelerate encoding module, we exploit similarity between features in consecutive inputs to eliminate the redundant computation during the encoding. Our observation on wide range of applications show that consecutive inputs have 78 percent average and up to 97 percent similarity. In our experiment we calculated the similarities between consecutive inputs of four different datasets. In these datasets most of the inputs are more than 40 percent similar to the previous input and only a few samples have less than 20 percent similarity(illustrated in Fig. 9). This similarity between consecutive inputs makes the encoded hypervectors similar in the high dimensional space as well. Therefore, instead of encoding each input to a hypervector, `HD-Core` generates the encoded hypervector of the current input by modifying the encoded hypervector of the previous input. This significantly reduces the required resources to generate each dimension of the encoded hypervector.

After encoding the input, the associative search module calculates the similarity metric between the encoded input and the HD model. In order to achieve acceptable accuracy on practical classification problems (i.e., speech, activity, or face recognition), HD computing has to use class hypervectors with non-binary elements [26]. The non-binary model, unlike the binary model that uses a simple *hamming distance* metric, uses a more complex *cosine* metric to find the similarity between a query hypervector and class hypervectors. The *cosine* can be calculated using the dot product of an input hypervector with all stored class hypervectors, which involves a large number of multiplication/addition operations. To improve the efficiency of the associative search step, `HD-Core`, by taking the statistical properties of the hypervector, employs a clustering algorithm to share the values in each class hypervector. Thus, instead of multiplying all pairs of the query and the class hypervector, `HD-Core` adds all query elements which are going to multiply with a shared class element and finally multiplies the result of addition with the corresponding class value.

The encoding and associative search stages of HD computing consist of a substantial number of binary/fixed-point addition and multiplication operations. HD computing operations can be deeply pipelined and parallelized at dimension level. These inherent characteristics of HD computing make FPGAs, that can provide flexibility in design and huge parallelism [13] with high energy efficiency [27], [28], an excellent match for implementing HD computing applications. In this paper, we propose novel techniques to exploit computation reuse for HD computing (`HD-Core`). `HD-Core` reduces the computation complexity of the encoding module by reusing the previously encoded hypervector. It also increases the performance of the associative search by replacing the multiplication operations with addition operations by clustering and sharing the values of class hypervector elements [29]. We also proposed an FPGA-based Acceleration of `HD-Core`, which significantly reduces the computational cost of both encoding and associative search.

We evaluate the impact of the `HD-Core` optimizations on the efficiency of the wide range of classification applications. Our evaluations on a wide range of classification problems show that `HD-Core` encoding provides $5.7\times$ $(2.3\times)$ energy efficiency, and $4.5\times$ $(2.1\times)$ speedup as compared to GPU (state-of-the-art FPGA [13]) implementation. `HD-Core` also provides $4.4\times$ $(1.4\times)$ energy efficiency improvement and $4.8\times$ $(2.4\times)$ speedup as compared to GPU (state-of-the-art FPGA) implementation while ensuring the same quality of classification. We observe that 40 percent of `HD-Core` performance improvement comes directly from the encoding acceleration and the rest is coming from our previous optimization that accelerated the associative search [29].

2 Background and Related Work

2.1 HD Computing Algorithm

HD provides a general model of computing which can apply to various types of learning problems. Classification is one of the most commonly used learning algorithms. Fig. 1 shows the overall structure of the HD classification in both training and inference phases. Encoding module maps input data to a hypervector, a vector with D_{hv} dimensions. Training is performed on hypervectors by adding all hypervectors corresponding to a particular class together. During training, a single hypervector is generated for each existing class. These class hypervectors are stored as the *HD model*. HD computing, during the inference, consists of two main steps: encoding and associative search. It uses the same encoding scheme to map a query to a hypervector with D_{hv} dimensions. Finally, on the associative search step, it performs the reasoning task by searching for a class hypervector, which has the highest similarity to the input hypervector.

2.2 Encoding

The first step in both training and inference of HD computing is encoding the input data to a hypervector. The main goal of encoding module is to map the input data to a hypervector with D_{hv} dimensions (e.g., $D_{hv} = 10,000$), while keeping important information of a data point in the original space, e.g., the feature values and their indexes in the input feature vector. Based on the input data representation and the hardware platform, various encoding approaches have been introduced in the literature [12], [19], [30]. In this paper, we focus on a method from [31] since it requires less memory to store the base hypervectors and is more FPGA-friendly. The encoding is performed in three steps.

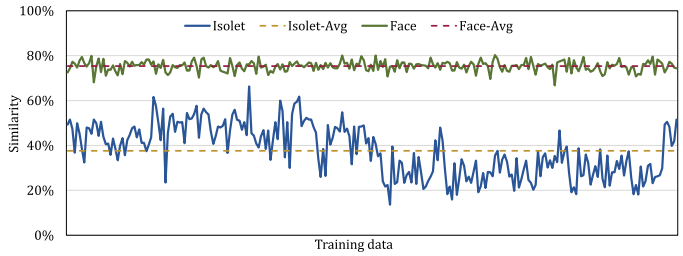

Fig. 2. Similarity between consecutive inputs in two datasets.

1) *Base hypervector generation*: First, the values of features based on the distribution of the feature values are quantized to L levels [31], [32]. Then, each level is assigned to a pre-generated hypervector, called base hypervector. To generate the base hypervector for feature level with the minimum value, BHV_0 a random binary hypervector with D_{hv} dimensions is generated. To generate the base hypervector representing the maximum value of the features, BHV_L, $D_{hv}/2$ dimensions of BHV_0 are selected and flipped randomly to produce an orthogonal base hypervector to BHV_0. To generate BHV_l for feature with level l, $\frac{D_{hv}/2}{l1}$ dimensions of the previous base hypervector, starting from the initial base hypervector, are flipped. As a result, features with closer values have more similar base hypervectors, while the minimum and maximum level of the features will are nearly orthogonal. Base hypervectors are generated offline and stored in the encoder module.

2) *Base hypervector permutation*: After generating the base hypervectors, each element ($Feature_i$) of a given input feature vector is mapped to its corresponding base hypervector $BHV(Feature_i)$. To take the spatial position of input features into account, the encoding module uses a permutation operation $P^i(BHV)$, where i is the index of the feature in the input feature vector. $P^i(BHV)$ can be an i-bit left rotational shift on the base hypervector. Since base hypervectors have large dimension, and they are randomly generated, permutation generates an orthogonal hypervector to its resultant shift orthogonal.

3) *Base hypervectors aggregation*: Eventually, the permuted hypervectors are aggregated to generate the encoded hypervector. Each dimension of the encoded hypervector is generated by adding the corresponding dimension of all the permuted hypervectors together. Equation (1) represents the encoding function. For all the input features, we first read their corresponding base hypervectors, $BHV(Feature_i)$. Then each hypervector is permuted based on its location in the input feature vector , $P^i(BHV(Feature_i))$. Then we perform an element-wise addition on all the permuted hypervectors

$$Encoding(input) = \sum_{i=0}^{\#Features} P^i(BHV(Feature_i)). \quad (1)$$

2.3 HD Model Training

HD computing performs the training procedure on the encoded hypervectors. For all data corresponding to a particular class, HD computing adds all hypervectors element-wise to create a class hypervector. For example, assume $Q^i = \{q_1^i, q_2^i, \ldots, q_D^i\}$ is a hypervector belongs to the class ith. As shown in Equation (2), the HD model learns the common patterns and/or features of a class by adding all the encoded hypervectors of training data with the same class tag. To train the HD model, we first encode, all the inputs belongs to the same class and perform an element-wise addition on the encoded hypervectors to achieve a hypervector representing the class

$$\mathbf{C}^i = \{w_1^i, w_2^i, \ldots, w_D^i\} = \sum_j Q_j^i. \quad (2)$$

Elements of the class hypervectors can have non-binary values. Non-binary hypervectors significantly increases the inference cost, as the rest of the reasoning task, i.e., similarity check, will be integer operations instead of binary operations. To reduce the computational cost, several prior works tried to binarize the class elements after training by applying a *majority* function on each dimension [33], [34]. However, these techniques lose some of the information stored in each class hypervector, thereby scarificing the accuracy forthe performance.

2.4 Associative Search

After training, all class hypervectors are stored as the HD model (shown in Fig. 1). In inference, an input data is encoded to the *query hypervector* using the same encoding module used for training. The associative search module is responsible for comparing the similarity of the input query hypervector with all stored class hypervectors and selecting a class with the highest similarity. Associative search module can use different similarity metrics to find a class which has the most similarity to a query hypervector. For class hypervectors with binarized elements, *Hamming distance* is an inexpensive and suitable similarity metric, while class hypervectors with non-binary elements need to use *cosine* as the similarity metric. Most existing HD computing techniques are using binarized class hypervectors in order to eliminate the costly *cosine* metric [31], [33]. However, it has been shown that HD with binary model provides lower classification accuracy as compared to the non-binary model [26].

2.5 Hardware Acceleration

HD comprise numerous but simple operations due to is high-dimensional nature. Prior work has proposed both algorithmic innovations as well as hardware accelerators to accelerate HD computing. The works [33], [35], [36], [37] proposed processing in-memory platforms, and the works [38], [39] proposed ASIC accelerators to run binarized HD models. The work in [36] fabricated a 3D VRRAM/CMOS to support the primary operations of HD computing (multiplication, addition, and permutation) on 4-layer 3D VRRAM/FinFET. The work in [33] focuses on accelerating binarized HD model using digital, they design digital, resistive, and analog associative memories to accelerate Hamming distance similarity in HD inference.

FPGAs provide high parallelism that can significantly improve the performance and energy efficiency of HD computing. Moreover, FPGA-based accelerators are advantageous over more specialized platforms, ASIC or processing in-memory, as they allow more flexibility in the HD model and easier customization of the model parameters such as

the length of the hypervectors, the precision of the HD model (binary or non-binary) and the input features characteristics. The works [13], [29], [40], [41] use FPGAs to accelerate HD computing. The study in [40] proposes a synthesizable VHDL library for training and inference of HD on FPGAs. The accelerator is limited to the binarized model and it uses logical operations to generate the base hypervectors during the runtime to reduce the costly memory accesses to read the base hypervectors. The authors, additionally, propose approximate logics to compose the binary class hypervectors without requiring to hold the summation on hypervector components in a multi-bit format during the model training. The work in [13] proposed an automated tool to generate FPGA-based HD accelerator. Work in [13] supports HD training, retraining, and inference phases, and supports accelerators for HD model with different quantizations (power-of-two, binary, and fixed-point). However, the inference of HD computing is still time and energy consuming. To reduce the time and energy consumption of HD inference, we exploited computational reuse methods to reduce the computation complexity of HD inference. As we proposed in our previous work [29], FACH represents class elements of a trained HD model using clustering algorithms. During runtime, instead of multiplying all inputs and class elements, FACH adds all the inputs belonging to the same class cluster centroid and multiplies the result once at the end. FACH assumes the encoded hypervector is stored in FPGA BRAM. However, in our experiments, we observe that the original encoding m odule, on average, takes 55 percent of the resources. Many of the previous works either assumed that the encoded hypervector is available in the memory [29], [33], [42] or they implemented the original encoding module [13], [38], [43]. In this work, we exploit the data locality between consecutive input data to significantly reduce the complexity of the encoding module and thereby accelerating the encoding module.

3 Proposed HD-Core

In HD computing, the first step in either training or inference is encoding the input feature vector, $\vec{V} = \{V^0, V^1, \ldots, V^F\}$, where F is the number of input features. The encoding uses basic permutation and binary addition on the base hypervectors to encode the input feature vector to a hypervector. Each possible value of the input feature vector, $\{l_0, l_1, \ldots, l_l\}$, has a corresponding element in the set of base hypervectors $\{B\vec{H}V_0, B\vec{H}V_1, \ldots, B\vec{H}V_l\}$. To generate the encoded hypervector, we need to read the base hypervector for each feature and permute it. All the permuted hypervectors are aggregated to generate the encoded hypervector. For unchanged features between consecutive inputs, the permuted hypervectors remain unchanged therefore, it can be reused for generating the encoded hypervector of the current input.

By quantizing the value of the feature to L levels, consecutive inputs share more features with the same value. For unchanged features in consecutive inputs, the permuted hypervectors will have the same contribution in building the encoded hypervector for both current and previous input. Thus, higher similarity between consecutive inputs results in higher efficiency of the HD-Core encoding. Figure

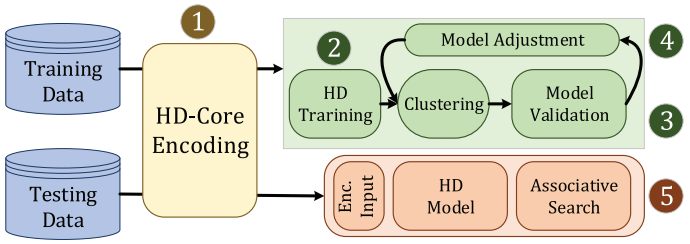

Fig. 3. HD-Core supporting Framework consisting of HD encoding, training and inference.

shows the similarity of consecutive inputs in the training data for speech recognition (ISOLET [44]) and face detection (FACE [45]) dataset. The similarity in the test data would follow the same trend since the training data was shuffled for the sake of generalization. Although consecutive inputs in the ISOLET dataset have shown on average 36 percent similarity, they can be upto 67 percent similar. This metric for the FACE dataset is considerably higher, consecutive inputs share 79 percent of their features on average. By quantizing the value of the feature to L levels, consecutive inputs share more features with the same value. HD-Core leverages the high similarity between consecutive inputs, to reduce the computation complexity of the encoding step.

HD-Core exploits the statistical characteristic of the HD computing in order to reduce the HD computational complexity. First, HD encodes all data points to hypervectors and send it to the training and inference phases. Fig. 3 shows an overview of the HD-Core framework consisting of five main steps: encoding, training, model quantization and validation, model refinement, and inference. After encoding the inputs(❶), HD-Core trains the HD model by combining data points corresponding to each class (❷). HD refinement clusters the values that elements in each class hypervector can take by applying non-linear clustering on the trained class hypervectors. This method reduces the possible values that the elements of each class hypervector can take. Also, HD refinement estimates the accuracy of the new HD model on the validation data, which is a part of the training data (❸). If the error rate is more significant than a pre-defined ϵ value, HD-Core adjusts the model and again clusters all values exist in each newly trained class hypervector. This clustering gives us new centroids, which better represent the distribution of the values in each class hypervector. This process continues iteratively until the convergence condition ($\Delta E < \epsilon$) is satisfied, or the algorithm has run for a pre-defined number of iterations (❹). When the convergence condition satisfied, HD-Core framework sends a new HD model with the clustered class elements to inference in order to perform the rest of the classification task. HD-Core uses the modified HD model with clustered class elements for inference (❺). In the following subsections we explain the HD-Core encoding and associative search in detail.

3.1 HD-Core Encoding

To reduce the complexity of encoding and reduce the required resources, we exploit locality to generate the currently encoded hypervector by reusing the encoded

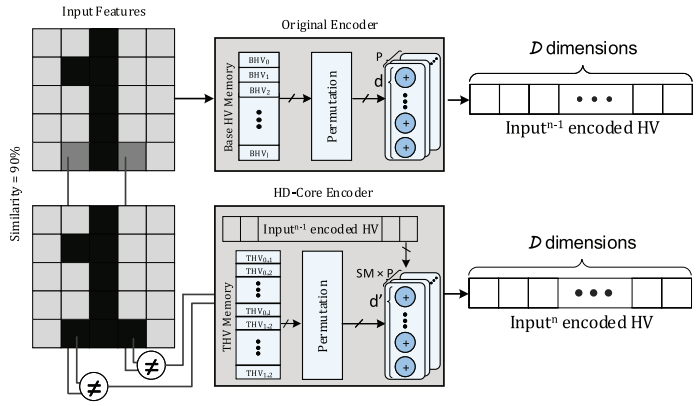

Fig. 4. Original and HD-Core encoding.

hypervector of the previous input. The encoded hypervector is the result of aggregating the permuted hypervectors over different features. The unchanged features have the same permuted hypervector while for the other features a new permuted hypervector is needed. To reuse the previously encoded hypervector, the permuted hypervector of the feature that changed from the previous input should be subtracted from the encoded hypervector. Then, the permuted hypervector of the current feature should be added to the encoded hypervector. Therefore, the operations for unchanged features are reused from the previous input. However, for changed features, one subtraction and one addition are required to generate every dimension of the encoded hypervector. Therefore, each change in the input feature vector has one operation overhead.

This technique reduces the number of required operations whenever the similarity between consecutive inputs is higher than 50 percent. Having higher than 50 percent similarity between consecutive inputs is a strict condition which is not practical in all applications. To eliminate the additional operation overhead, HD-Core introduces transition hypervectors (THV), which are hypervectors that represent the transition between one feature level to another feature level. A transition hypervector is the base hypervector for a feature level subtracted by the base hypervector of the previous feature level. $L-1$ transitions are possible for each level of feature. For level i of feature, the transition from level i to {level $0, \ldots,$ level $i-1$} and to {level $i+1, \ldots$, level l} are possible. Equation (3) shows the transition hypervector when the value of a feature changes from level i to level i'

$$\begin{aligned} &\mathrm{THV}_{i\rightarrow i'} = BHV_{i'} - BHV_i \\ &THV_{i'\rightarrow i} = BHV_i - BHV_{i'} \Rightarrow THV_{i'\rightarrow i} = -THV_{i\rightarrow i'}. \end{aligned} \tag{3}$$

According to the Equation (3), the transition hypervectors for transition from i to i' is the negative form of that for transition from level i' to level i. HD-Core, instead of storing all the transition hypervectors, only stores the transition hypervector for the transitions from a lower level to a higher one. To generate transition hypervectors to/from the level 0 from/to any other $L-1$ levels, base hypervector of level 0 is subtracted from the base hypervector of level i to generate transition hypervector $THV_{0\rightarrow i}$. Equation (4) shows all the transition hypervectors that we store in the memory which includes transitions from level i to the higher levels. The rest of the transition hypervectors are generated and stored as $THV_{i'\rightarrow i}$ ($\forall i' < i$) which are the negative form of $THV_{i\rightarrow i'}$. In total, $\binom{L}{2}$ transition hypervectors are stored while in the original HD encoding L base hypervectors are stored. Therefore, as shown in Equation (5), $L \times \frac{L-3}{2}$ more hypervectors as the regular HD model are needed to store in the memory

$$THV_{i\rightarrow i'} \forall j \,|\, L \geq j > i \tag{4}$$

$$\frac{L \times L-1}{2} - L = L \times \frac{L-3}{2}. \tag{5}$$

Quantizing the input features to L levels not only increases the consecutive inputs similarity, but it reduces the the memory overhead of using the HD-Core encoding as well. Using the HD-Core encoding reduces the number of required operations proportional to the similarity of consecutive inputs. Unlike the original encoding, the number of required operations is not fixed; it depends on the type application and sequence of inputs. By pre-processing the training data, a metric for similarity (*SM*) of the consecutive inputs is calculated. Although the *SM* is calculated based on the characteristics of the training data, the same pattern is observed during the inference. *SM* represents the similarity of all two consecutive inputs with a value. The similarity between each two consecutive input can be: (i) less than the *SM*, (ii) equal to the *SM*, and (iii) higher than the *SM*. In the first case, the encoding cannot be done in one step. Extra steps are required to encode the input; in these steps, the associative search module will be stalled. In the second case, the encoding hardware is fully utilized, and the input is encoded in one step. In the case of having a higher similarity than *SM*, the underutilized encoding module can encode the input in one step. The function $Enc_L(d_i, d_{i-1}, SM)$ returns the number of required cycles to encode input d_i while the encoded hypervector of input d_{i-1} is available. The $SM\%$ is calculated based on the similarity to minimize the operations, based on the similarity of the consecutive inputs of training data. Calculating the optimum *SM*, additionally, depends on the hardware implementation. HD-Core uses the average similarity of the consecutive inputs in the training data for the *SM* parameter.

3.2 HD-Core Associative Search

Performing *cosine* similarity between two vectors involves calculating the dot product of vectors divided by the size of each vector. Since HD trains the model offline, the normalization of the class hypervectors can be performed offline. On the other hand, input data is common between all class hypervectors, thus it does not need to be normalized. Therefore, *cosine* similarity between a query $Q = \{q_1, q_2, \ldots, q_D\}$ and ith class hypervector, $C^i = \{w_1^i, w_2^i, \ldots, w_D^i\}$, requires calculating their dot product which involves D additions and D multiplications, where D is the dimension of the hypervectors.

In this work, model refinement in HD-Core reduces the class span by carefully selecting a subset from the input spaces, called "best representatives". HD-Core limits the number of values that each class element can take (i.e., $\{w_1, \ldots, w_D\} \in \{c_1, \ldots, c_k\}$ and $k << D$)). This enables us to remove the majority of *cosine* multiplications by

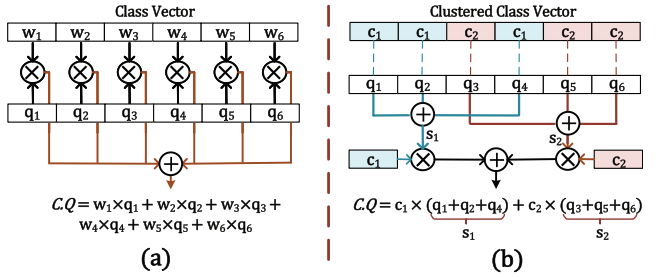

Fig. 5. An example of dot product between the class and query vectors with six dimensions (a) using conventional method, (b) when the elements of the class vector clustered.

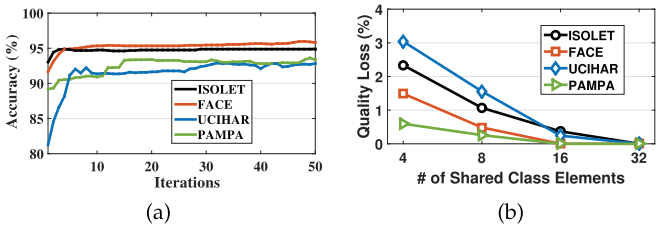

Fig. 6. a) The classification accuracy of applications during retraining iterations. b) Impact of number of class elements on the quality loss of different applications.

factorization. In other words, instead of multiplying the D elements of query and class hypervector, we add the input data for all dimensions for which class hypervector has the same element. Finally, the result of the addition is multiplied by the value of that particular class.

Here we explain how HD-Core can limit the number of each class elements with no or minor impact of classification accuracy. To find representative class elements, the clustering algorithm is applied to the pre-trained class hypervectors. For each class hypervector, our design identifies a specified number of clusters, say k, based on clustering algorithms. The centroids of clusters are selected as the representative weights and stored into the weight table. Assuming that the actual numerical values belong to a set θ, the objective of the clustering algorithm is to find a set of k cluster centroids $\{c_1,\ c_2,\ \ldots,\ c_k\}$ that can best represent the class values ($c \in N$)

$$\{w_1^i, w_2^i, \ldots, w_D^i\} \in \{c_1^i, c_2^i, \ldots, c_k^i\}. \tag{6}$$

Formally, the objective is to reduce the Within Cluster Sum of Squares (WCSS)

$$\min_{c_1,\ c_2,\ \ldots, c_k} \left(WCSS = \sum_{j=1}^{k} \sum_{\theta_i \in c_j} ||\theta_i - c_j||^2 \right), \tag{7}$$

where θ_i is the ith sample drawn from θ and k is the number of clusters.

We use the k-means clustering algorithm to solve the minimization objective for each HD class hypervector separately, as the distribution of values can vary across different classes. The calculation of dot product between query, Q, and a class hypervector, C^i, can be simplified by adding all query elements which belong to the same cluster in class hypervector. For example, for class dimensions with c_k elements, our design adds all corresponding query elements together ($s_k = \sum_j = q_j$ where $w_j = c_k$). In a similar way, our design calculates the accumulative query elements on all k cluster centroids: $\{s_1, s_2, \ldots, s_k\}$ and $s \in N$. Finally, these values multiply with each corresponding cluster values and accumulate together to generate a dot product between Q and C^i hypervectors

$$Q.C^i = s_1 \times c_1 + s_2 \times c_2, + \ldots s_k \times c_k. \tag{8}$$

This method reduces the number of multiplications involved in dot product from D to k, where k can be about three orders of magnitudes smaller than D. Fig. 5 shows an example of the dot product between a class and a query vector using conventional method and clustered model. Since in the conventional method, the class elements can take any value, the dot product involves six multiplications (Fig. 5a). HD-Core exploits the advantage of clustered class values to first add the query elements corresponding to the same centroid and then multiply the result with the centroid values (Fig. 5b). This reduces the number of multiplications to two.

Error Estimation. Sharing the elements of input and class hypervectors reduces the HD classification accuracy. After the training, our design replaces the elements of the class hypervectors with the closest representative values (cluster centroids). We estimate the error rate of the new model by cross-validating the cluster HD on a validation data, which is a part of the training data. The quality loss, ΔE is defined as the error rate difference between the HD using original and modified models ($\Delta E = E_{clustered} - E_{original}$).

Model Adjustment. If the error rate does not satisfy the tolerance $\Delta E < \epsilon$, HD-Core adjusts the new model by retraining the network over the same training dataset. In retraining process, HD composer looks at the similarity of each input hypervector to all stored class hypervectors; (i) if an input data correctly matches with the corresponding class in associative memory, our design does not change the mode. (ii) if an input hypervector, Q, wrongly matches with the ith class hypervector (C^i) while it actually belongs to jth class (C^j), our retraining procedure subtracts the input hypervector from the ith class and add it to jth class ($\overline{C^i} = C^i - Q$ & $\overline{C^j} = C^j + Q$). After adjusting the model over the training data, HD refinement again clusters the data in each class hypervector and estimate the classification error rate. We expect the model retrained under the modified condition to better fit with the clustered values. Since we start clustering from first iterations, both baseline and HD-Core show almost the same pattern in increasing the accuracy of the model during the retraining. If an error criterion is not satisfied, we follow the same procedure until an error rate, ϵ, is satisfied or we reach to a pre-specified number of iterations. After the iterations, the new model, which is compatible with the proposed accelerator, is used for real-time inference.

Fig. 6a shows the classification accuracy of applications during different retraining iterations when the class elements are clustered to 32 values. Our evaluation shows that HD refinement can compensate for quality loss due to clustering by using less than 30 iterations. Since the maximum number of retraining iterations is limited to 30 and the baseline HD model also requires the refinement iterations, the overhead of the HD-Core training is negligible. All pre-processing operations in the HD refinement module are

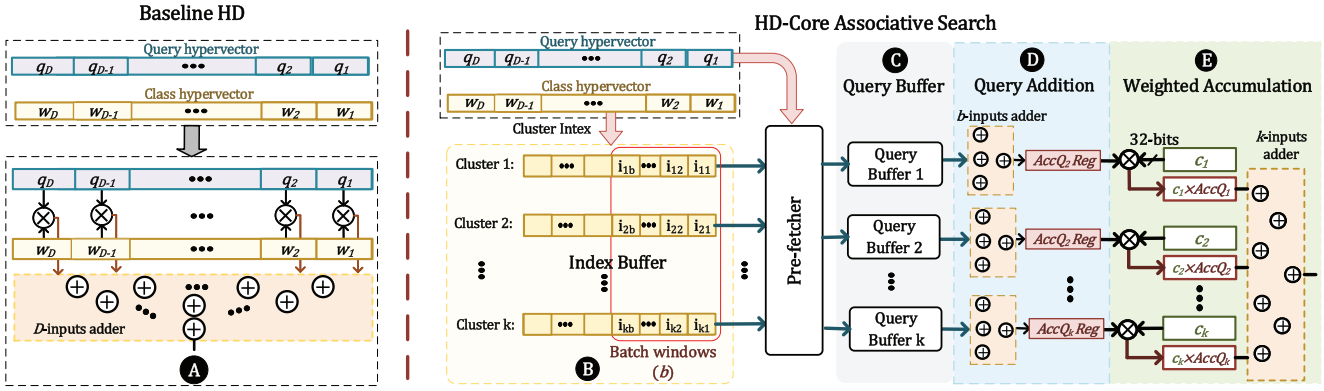

Fig. 7. FPGA-based implementation to calculate the dot product between a query and class hypervectors in baseline HD and HD-Core with clustered class elements.

performed offline and their overhead is amortized among all future executions of HD-Core accelerator. Fig. 6b shows the final quality loss, ΔE, when HD-Core clusters the class hypervector to a different number of centroids. We consider the cluster sizes of 4, 8, 16 and 32. The results show that different applications can provide $\Delta E = 0\%$ while using a different number of class clusters. For example, face recognition can achieve $\Delta E = 0\%$ when the class elements are clustered to 16 centroids, while human activity recognition (UCIHAR) achieves $\Delta E = 0\%$ using 32 cluster centroids. In Section 5, we will explain the accuracy-efficiency trade-off in HD-Core using different clusters.

4 HD-Core FPGA Acceleration

We use FPGA to accelerate HD computing encoding and inference. In this we implement the HD-Core on an FPGA and use the platform proposed in [13] as the baseline FPGA implementation. Fig. 7A shows that the FPGA-based implementation of the baseline HD requires multiplication for S parallel dimensions to calculate the dot product between the query and class hypervectors. Then, the results of all S multiplications accumulate in a tree-based adder. The size of the S, the number of input dimensions which FPGA reads at a time, depends on the number of classes, and the number of available resources (e.g., LUTs, BRAM, and DSPs) in FPGA. In this case, our design sequentially generates the first S elements of the query vector and multiply it to the corresponding class elements ($S < D$). Then, the computations on the rest of the query elements are performed sequentially. In the following, we explain how each design can be accelerated on FPGA.

4.1 HD-Core Encoding Acceleration

The first step in HD is encoding the input feature vector $\vec{v}$ into the query hypervector $\vec{Q}$, using fundamental permutation and addition on the transition hypervectors (the baseline encoding instead of using transitions hypervectors, uses base hypervectors). As previously shown in Section 3.1, for input features that are not changed from the previous input no additional operation is needed, while for the others, the transition hypervector should be either added or subtracted from the previously encoded hypervector. Recall that the dimensions of transition hypervectors binary numbers that aggregate in a dimensionwise pattern to generate each dimension of the encoded hypervector. Dimensions of the encoded hypervector can be in various widths and representations (e.g., fixed-point, binary, etc.). HD computing can be parallelized at dimension level since the similarity metric for each generated dimensions of the encoded hypervector can be calculated independent of the other dimensions. The final similarity metric is the aggregation of dimensions similarity. The encoding and associative search blocks are working in a pipeline structure; therefore, to maximize the resource utilization, the number of encoded dimensions generated in each cycle should be equal to the number of dimensions that the associative search module can process. Therefore, we segregate the encoded hypervector into the segments of S dimensions whereby at each clock cycle, one segment is generated. The generated S dimensions are passed to the associative search module to calculate the similarity metrics between the query hypervector and the class hypervectors. Thus, processing the entire query hypervector takes $T = \frac{D}{S}$ cycles.

The value of S, which represents the parallelism level, is limited by the available resources. R_E shows the required resources to generate a dimension of the encoded hypervector. $R_{A.S}$ shows the required resources for the associative search module to process a dimension of the encoded hypervector. Therefore $S \times (R_{Enc} + R_{A.S})$ should be less than the available resources of the FPGA. Since the number of operations executed in each cycle in the HD-Core encoding as compared to the baseline encoding is reduced by $SM\%$, the required resources to implement the HD-Core encoding module is also $SM\%$ less than the baseline $R_{HD-Core-Enc} = SM\% \times R_E$. These resources dedicated to the HD-Core encoding module can generate S dimensions of the encoded hypervector when the similarity of current input and the previous input is higher than or equal to SM. In general, Equation (9) shows the number of iterations required to generate S dimensions of the encoded hypervector, while $R_{HD-Core-Enc}$ resources are dedicated to the encoding module

$$\text{Latency}(\mathrm{d}_i, d_{i-1}, SM) = \left\lceil \frac{1 - Similarity(d_{i-1}, d_i)}{SM} \right\rceil. \quad (9)$$

4.2 HD-Core Associative Search Acceleration

Fig. 7 illustrates the HD-Core architecture, which supports dot product between a query and a single class hypervector. The class hypervector has k clustered values, i.e., the class elements can take one of the k cluster centroids, $\{c_1, c_2, \ldots, c_k\}$. To accelerate HD-Core, our design creates k index buffers,

where each buffer represents one of the cluster centroids (Ⓑ). Each buffer stores the indices of the class elements which have clustered to the same value. For example, the first index buffer, shown in Fig. 7B, stores all class indices which have the value as c_1. Since each class has D dimensions, we require $log_2\mathbf{D}$ bits to store each index.

Due to resource limitation of FPGA, we can only read S dimensions of the query hypervector at a time and process the remaining dimensions in sequential windows. However, sequentially accessing the query elements increases the number of resource requirements, since all S elements in a read window might belong to any of the clusters. In this case, each index buffer requires a tree-based adder with S inputs in order to take care of the worst case scenario, when all S query dimensions correspond to a single cluster. Instead, in this work, each read window accesses to $b = S/k$ indices from each index buffer. This method ensures that the number of required resources to add the element of each index buffer is less than b. We define this b window size as the batch size. In order to speed up the computation, HD-Core stores the index buffers, which are a compressed/trained HD model, inside the FPGA. These buffers are implemented using distributed memory using LookUp Table (LUT) and Flip-Flop (FF) blocks.

Each element of the index buffer points to one dimension of the query hypervector. In order to maximize the FPGA resource utilization, for all elements of index buffer in a batch window, HD-Core pre-fetches the query elements and store them in query buffers (Ⓒ). Next to each query buffer, a tree-based adder accumulates all S/k indices corresponding to a particular centroid (Ⓓ). The results of these additions are stored in registers. Next, FPGA processes the next batch sequentially. HD-Core is implemented in a pipeline, where the pre-fetching of the elements to query buffer performs simultaneously with the addition of the query elements which have been pre-fetched to query buffers in the last iteration. This pipeline can perform very efficiently since these two tasks require different types of FPGA resources. The indexing and pre-fetching are memory-intensive tasks and mostly utilize BRAM, LUTs, and FFs, while the addition of query elements mostly utilizes DSPs.

After every iteration, the values corresponding to the registers are accumulated. Once HD-Core has processed all D dimensions of the hypervector, each register has the accumulated query elements in all the dimensions for which class hypervector has the same clustered value. For each index buffer, our design multiplies the value of the register with the corresponding cluster value. The results of multiplication for all cluster centroids are then accumulated in order to generate the final dot product (Ⓔ). Regardless of the method used for calculating dot product, our design needs to compare the dot products for all existing classes and select the class which has the maximum similarity with the input vector.

5 Results

5.1 Experimental Setup

The proposed HD-Core has been implemented with software and hardware modules. For software support, we use Python to find the similarity metric between consecutive inputs in the training data. To eliminate the dependency of the *SM* to the distribution of the training data, we shuffle the training data for 10 times and calculate the average similarity each time and use the average as the *SM*. We exploit Scikit-learn library [46] for clustering and C++ software implementation for the HD model training and verification. For hardware support, we use FPGA to accelerate HD computing. We fully implement HD-Core inference functionality in RTL using Verilog HDL. HD-Core is deeply pipelined to run with 200 MHz clock frequency. We verify the timing and functionality of the design using both synthesis and real implementation of the HD-Core using Xilinx Vivado Design Suite [47]. To estimate the power consumption of the FPGA we use the builtin Xilinx Power Estimation tool in Vivado Desigh suite. HD-Core is implemented on the Kintex-7 FPGA KC705 Evaluation Kit. We compare the performance and energy efficiency of the FPGA-based implementation of HD-Core with the NVIDIA GTX 1080 GPU. The GPU-based implementation uses the same algorithmic optimization for the associative search as HD-Core; however, to maximize the performance of the GPU-based implementation, it uses the baseline encoding since we observed it provides a higher performance on GPU. The performance and energy of GPU are measured by the nvidia-smi tool. We also compare the HD-Core implementation with the state-of-the-art FPGA-based accelerator proposed in [13] as the FPGA baseline.

5.2 Workloads

We evaluate the efficiency of the proposed HD-Core on four popular classification applications, as listed below:

Speech Recognition (ISOLET). The goal is to recognize voice audio of the 26 letters of the English alphabet. The training and testing datasets are taken from ISOLET dataset [44].

Face Recognition (FACE): We exploit Caltech dataset of 10,000 web faces [45]. Negative training images, i.e., non-face images, are selected from CIFAR-100 and Pascal VOS 2012 datasets [48].

Activity Recognition (UCIHAR) [49]: The dataset includes signals collected from motion sensors for 8 subjects performing 19 different activities. The objective is to recognize the class of human activities.

Physical Activity Monitoring (PAMAP) [50]. This dataset includes logs of 8 users and three 3D accelerometers positioned on arm, chest and ankle. They were collected over different human activities such as lying, walking and, ascending stairs, and each of them corresponded to an activity ID. The goal is to recognize 12 different activities.

5.3 HD-Core Encoding

The computation complexity of the encoding step increases with the number of input features. In both training and inference, encoding consumes a great portion of resources. Fig. 8 shows the ratio of the LUT utilization in the encoding module to the LUT utilization of the entire accelerator. We compared the resource utilization of the baseline encoding with the HD-Core encoding when the associative search module is the HD-Core with 4 and 8 centroids (C4 and C8 respectively). In the ISOLET dataset, due to its higher number of classes, the complexity of the associative search is

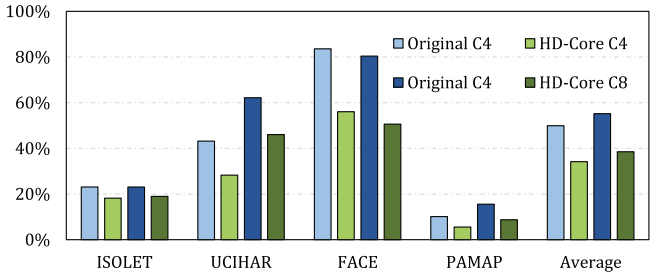

Fig. 8. LUT utilization ratio of the encoding module to the entire accelerator for the baseline encoding and HD-Core encoding with associative search modules with 4 and 8 centroids.

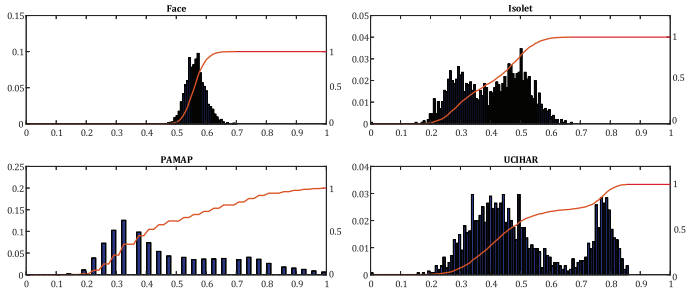

Fig. 9. Histogram and CDF of similarities between consecutive inputs for each dataset.

higher. Therefore, the associative search in ISOLET requires more resources to calculate the similarity metric for a dimensions than other applications. As shown in the figure, the baseline encoding module consumes 23 percent of the entire LUT resources, while the proposed HD-Core encoding module drops the LUT utilization to 18 percent. In the FACE dataset, most of the resources are dedicated to the encoding module as there are only 2 classes in the dataset. Therefore, the baseline encoding module consumes 82 percent of the entire resources. Nevertheless, by using the HD-Core encoding, only 54 percent of the resources are dedicated to the encoding module. Note that, HD-Core encoding not only utilizes less LUTs than the baseline encoding module, it generates more dimensions ($2.1\times$) of the encoded hypervector per cycle than the baseline encoding. In PAMAP, since the number of input features are relatively low, the complexity of the encoding is significantly less than the other datasets.

We implement the HD-Core encoding on FPGA and exploit input similarity to accelerate the encoding step. As explained in Section 3.1, quantizing the value of the input features increases the similarity between consecutive inputs. However, quantization may drop the classification accuracy. Table 1 shows the impact of feature quantization on the classification accuracy as well as the average similarity between consecutive inputs. As illustrated in Table 1, for all of the datasets, accuracy starts dropping when the input features quantized to 2 bits. However, the classification accuracy for FACE drops 0.2 percent when the input features are quantized to 2 bits. In the rest of the paper, we use the minimum bit width for the input features that provides the maximum accuracy. Thus, in in our experiments, we quantize the input features to 2 bits for FACE application, and to 3 bits for the rest of the applications. In Fig. 9, blue bars show the histogram of the similarity between consecutive inputs in training data, and the red line shows the Cumulative Distribution of the similarities between consecutive inputs for each dataset. As illustrated in the figure, only a few of the inputs in all of the datasets have less than 20 percent similarity and most of the inputs have higher similarity than the *SM* mentioned in Table 1.

HD-Core encoding stores the transition hypervectors instead of base hypervectors; thus, BRAM usage in HD-Core encoding is higher than the baseline encoding. In Fig. 10, the blue line shows the required memory to store the transition hypervectors in MB (left axis) for different quantization levels, while the dashed red line shows the available BRAM in the Kintex FPGA. The green line in Fig. 10 shows the average similarity between consecutive inputs in all the datasets for different quantization levels. In our experiments we observed that quantizing the input features to even to 4 bits has no impact on the prediction accuracy, while observing 34.8 percent similarity between consecutive inputs in all the datasets on average. Due to the limitation of the FPGA BRAMs, HD-Core encoder can support inputs with features $2^5 = 32$ levels, where still inputs are on average 30.4 percent similar.

HD-Core encoding exploits the similarities between consecutive inputs to reuse the previously encoded hypervector, thereby reducing the required hardware to generate each dimension of the encoded hypervector. Fig. 11 compares the throughput and energy consumption of HD-Core encoding module with the GPU and FPGA baseline implementations. Throughput and energy consumption of HD-Core and the FPGA baseline are normalized to those of the GPU implementation. HD-Core encoding can encode $4.5\times$ and $2.1\times$ more inputs in a second as compared to GPU and FPGA baseline respectively. HD-Core encoding reduced the energy required to encode each input for $5.7\times$ and $2.3\times$ as compared to the GPU and FPGA baseline respectively. Fig. 11, additionally, shows that for applications with a higher similarity between inputs the throughput

TABLE 1
Impact of Input Quantization on the Average Similarity of Consecutive Inputs and the Classification Accuracy

	1 bit		2 bits		3 bits		4 bits	
	Acc(%)	SM(%)	Acc(%)	SM(%)	Acc(%)	SM(%)	Acc(%)	SM(%)
ISOLET	93.6	93	94.7	59	95.5	36	95.5	23
UCIHAR	95.9	99	97.3	75	98	51	98.1	36
FACE	89.9	99	92.9	79	93.1	55	93.1	38
PAMAP	94.3	72	95.9	56	96.7	44	96.8	39

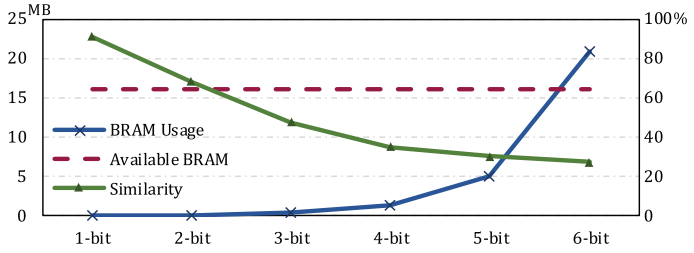

Fig. 10. HD-Core BRAM utilization and average similarity of consecutive inputs for different quantization levels of input features.

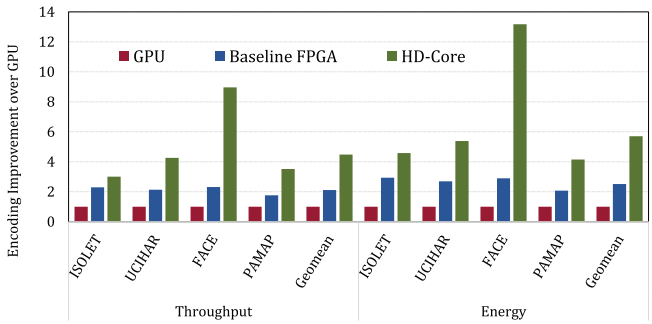

Fig. 11. Throughput and energy improvement of HD-Core encoding in comparison with GPU and the FPGA baseline [13].

improvement will be higher. In the FACE dataset which has the highest *SM* among the other datasets, HD-Core encodes $9\times$ more inputs per second with $13.2\times$ less energy as compared to the GPU implementation.

5.4 HD-Core Accuracy-Efficiency Trade-Off

Table 2 shows the throughput and the energy consumption of the HD-Core when it first encodes the data and then performs the associative search (Encoding and Associative Search), as well as when the encoded hypervector of inputs are stored in memory and HD-Core only performs the associative search. Throughput and energy consumption of HD-Core are compared with the baseline FPGA implementation. *C4*, *C8*, *C16*, *C32* show the number of shared elements (centroids) in each class hypervector. Associative Search column shows the impact of HD-Core when the encoded hypervector is stored in BRAM, and all of the FPGA resources are dedicated to the associative search module. Comparing the results of baseline HD with the HD-Core shows that HD-Core can improve the efficiency of the HD computing by reducing the number of operations in encoding and reducing the number of multiplication in the associative search step. HD-Core performance depends on the number of shared class elements. HD-Core with more number of centroids requires more FPGA resources to implement and therefore, its throughput is less than HD-Core with less number of centroids while providing a higher accuracy. The power consumption of the FPGA is highly correlated with the resource utilization and HD-Core with various number of centroids have close resource utilization and consequently close power consumption. Therefore, the energy consumption of the HD-Core with more centroids is higher since increasing the number of centroids reduces the throughput; consequently, increasing the execution time. As we discussed in Section 3.2, HD-Core accuracy depends on the number of shared class elements. Fig. 6b shows the impact of the number of centroids on the classification accuracy. As illustrated in this figure, as the number of centroids increases, the accuracy increases. For example, in UCIHAR, increasing the number of centroids from 4 to 32 increases the accuracy for 3 percent, at the cost of decreasing the performance and energy efficiency for 19, and 22 percent respectively.

HD-Core with 4 centroids, on average, shows $2.4\times$ and $1.4\times$ performance and energy improvement as compared to the baseline FPGA while dropping the accuracy for 2 percent. HD-Core encoding contributes to 40 percent of this performance improvement. Enabling the computation reuse in the encoding module significantly accelerates the design. HD-Core, by exploiting the computational reuse in the encoding module is 96 percent faster than HD-Core with the original encoding.

HD-Core with 32 centroids provides the same classification accuracy as the baseline HD while on average shows $1.8\times$ with 13 percent energy overhead. Comparing these results with the GPU-based approach shows that HD-Core with 32 centroids can provide $2.7\times$ higher energy efficiency and $3.6\times$ speedup. Performance and energy improvement can increase up to $4.8\times$ and $4.3\times$ by using 4 centroids. Fig. 12 compares the throughput and energy efficiency improvement normalized to those of the GPU. As illustrated in Fig. 12 the energy improvement for FACE dataset is significantly higher, while for PAMAP, the energy improvement is slightly higher than the FPGA baseline. In FACE dataset, since it has 2 classes, the required computation is less than the other datasets. Also, the similarity between consecutive inputs in this dataset is significantly higher than the other datasets. In PAMAP the number of input features is less than the other datasets; thus, the encoding will be significantly less complex. In this dataset, the associative memory is the bottleneck of performance since the encoding module requires fewer resources as compared to the others.

TABLE 2
Comparison of HD-Core With the FPGA Baseline Implementation in Terms of Throughput ($1000\times$ Classification Per Second) and Energy Consumption for Classifying an Input (mJule)

Dataset		Encoding and Associative Search					Associative Search				
		Baseline FPGA [13]	*C4*	*C8*	*C16*	*C32*	*Baseline FPGA [13]*	*C4*	*C8*	*C16*	*C32*
ISOLET	**Throughput**	258.5	621.1	565.0	537.6	507.6	560.0	621.1	565.0	537.6	507.6
	Energy	17.1	12.6	14.0	14.7	15.7	17.0	12.1	13.3	14.0	14.7
UCIHAR	**Throughput**	1119.4	2873.9	2816.9	2666.7	2020.2	1440.0	3076.9	2816.9	2666.7	2531.6
	Energy	3.5	3.2	3.1	3.3	3.9	3.2	2.4	2.6	2.8	2.9
FACE	**Throughput**	3348.2	7692.3	6666.7	5263.2	5000.0	4320.0	7692.3	6896.6	6451.6	6451.6
	Energy	3.2	1.2	1.3	1.5	1.7	1.9	1.0	1.1	1.1	1.2
PAMAP	**Throughput**	562.2	1242.2	1129.9	1075.3	1015.2	720.0	1242.2	1129.9	1075.3	1015.2
	Energy	5.8	5.7	5.8	6.1	6.6	7.0	6.0	6.3	6.8	7.2

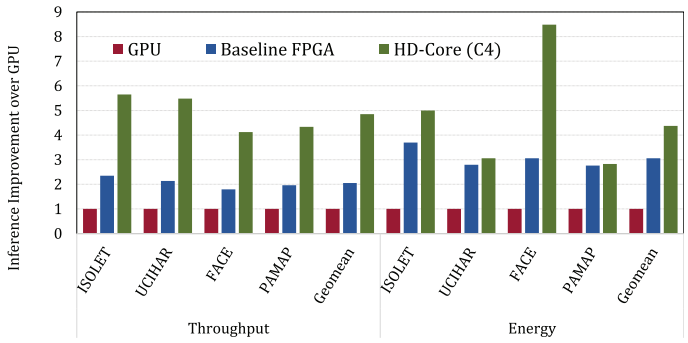

Fig. 12. Throughput and energy improvement in comparison with the GPU and FPGA baseline [13].

5.5 HD-Core Resource Utilization

Table 3 shows the resource utilization of the HD-Core as well as the encoding resource utilization breakdown for FACE and ISOLET datasets with 4, 8, 16, and 32 cluster centroids. For the HD-Core encoding, all the operations are implemented on LUTs, while the associative search module uses FPGA DSPs to calculate the similarity metric. HD-Core tries to fully utilize the FPGA resources; however, in our experiments, when the resource utilization is above 90 percent, design may face routing congestion issues. In HD-Core Encoding, BRAM utilization is independent of the number of centroids since BRAMs are only used to store the transition hypervectors. Due to the higher number of transition hypervectors in ISOLET, HD-Core encoding requires more BRAMs to implement ISOLET data encoding as compared to encoding the FACE dataset. Associative search module utilizes DSPs to calculate the similarity metric between the class hypervectors and the query hypervector. Therefore, as the number of classes increases, the number of operations in the associative search will also increase. Therefore, for ISOLET with 26 classes, since the number of operations in the associative search is much higher than the that in the FACE dataset with 2 classes, the number of DSPs limits the parallelism. While in the FACE dataset the encoding module is the bottleneck of parallelism. Increasing the number of centroids increases the HD-Core computation complexity; thus, more resources are required to provide the same parallelism. Since, the number of the available resources are fixed, increasing the number of clusters reduces the parallelism, which reduces the required resources to implement HD-Core encoding.

TABLE 3
Resource Utilization of HD-Core Encoding and HD-Core for FACE and ISOLET Datasets

Dataset	HD-Core	HD-Core Encoding Resource Utilization				HD-Core Resource Utilization			
		LUT(%)	FF(%)	BRAM(%)	DSP(%)	LUT(%)	FF(%)	BRAM(%)	DSP(%)
FACE	C4	74.8	1.4	2.7	0.0	89.5	5.8	18.7	36.8
	C8	70.7	1.2	2.7	0.0	86.7	3.3	9.4	31.8
	C16	69.2	1.3	2.7	0.0	84.9	3.4	6.1	30.2
	C32	73.2	1.3	2.7	0.0	90.2	5.1	5.8	30.3
ISOLET	C4	15.0	0.3	8.4	0.0	64.9	12.7	44.3	95.7
	C8	16.4	0.3	8.4	0.0	68.4	12.2	24.5	96.7
	C16	17.3	0.4	8.4	0.0	68.6	12.2	13.7	98.3
	C32	18.3	0.4	8.4	0.0	73.5	12.6	12.6	98.3

6 Conclusion

We propose a novel hyperdimensional computing FPGA implementation exploiting computation reuse, called HD-Core, which significantly reduces the cost of classification. The framework reuses previously encoded hypervector to reduce the complexity of the encoding step. It also extracts representative operands of a trained HD model using clustering algorithms. At runtime, instead of multiplying all inputs and class elements, our design adds all the inputs belonging to the same class cluster centroid and multiplies the result once in the end. Our evaluation over a wide range of applications shows that HD-Core can provide $4.8\times$ faster execution and $4.3\times$ higher energy efficiency as compared to the GPU implementation. The HD-Core encoding provides $2.1\times$ more throughput and $2.3\times$ energy efficiency as compared to the FPGA baseline implementation proposed in [13]. This efficiency in the encoding module contributes to 40 percent of the $2.4\times$ performance improvement of the HD-Core as compared to the state-of-the-art FPGA accelerator [13]. In future, we are going to apply HD-Core algorithmic optimizations on other hardware platforms including GPUs and ASIC.

Acknowledgments

This work was supported in part by CRISP, in part by one of six centers in JUMP, in part by an SRC program sponsored by DARPA, and in part by US National Science Foundation under Grant 1527034, Grant 1730158, Grant 1911095, and Grant 1826967.

References

[1] K. Das and R. N. Behera, "A survey on machine learning: Concept, algorithms and applications," *Int. J. Innovative Res. Comput. Commun. Eng.*, vol. 5, no. 2, pp. 1301–1309, 2017.
[2] W.-Y. Tsai *et al.*, "Always-on speech recognition using truenorth, a reconfigurable, neurosynaptic processor," *IEEE Trans. Comput.*, vol. 66, no. 6, pp. 996–1007, Jun. 2017.
[3] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis Lectures Hum. Lang. Technol.*, vol. 10, no. 1, pp. 1–309, 2017.
[4] R. Yazdani, J.-M. Arnau, and A. González, "A low-power, high-performance speech recognition accelerator," *IEEE Trans. Comput.*, vol. 68, no. 12, pp. 1817–1831, Dec. 2019.
[5] T. Luo *et al.*, "DaDianNao: A neural network supercomputer," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 73–88, Jan. 2017.
[6] Y. Liu, Y. Wang, F. Lombardi, and J. Han, "An energy-efficient stochastic computational deep belief network," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2018, pp. 1175–1178.
[7] S. Salamat, M. Imani, S. Gupta, and T. Rosing, "RNSnet: In-memory neural network acceleration using residue number system," in *Proc. IEEE Int. Conf. Rebooting Comput.*, 2018, pp. 1–12.
[8] A. M. Costoya, C. F. Frasser, M. Roca, and J. L. Rossello, "Energy-efficient pattern recognition hardware with elementary cellular automata," *IEEE Trans. Comput.*, vol. 69, no. 3, pp. 392–401, Mar. 2020.
[9] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "SparCE: Sparsity aware general-purpose core extensions to accelerate deep neural networks," *IEEE Trans. Comput.*, vol. 68, no. 6, pp. 912–925, Jun. 2019.
[10] A. Ardakani, C. Condo, and W. J. Gross, "Fast and efficient convolutional accelerator for edge computing," *IEEE Trans. Comput.*, vol. 69, no. 1, pp. 138–152, Jan. 2020.
[11] K. Seto, H. Nejatollahi, J. An, S. Kang, and N. Dutt, "Small memory footprint neural network accelerators," in *Proc. 20th Int. Symp. Qual. Electron. Des.*, 2019, pp. 253–258.
[12] 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 *Proc. 56th Annu. Des. Autom. Conf.*, 2019, Art. no. 52.

[13] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-HD: Fast flexible FPGA-based framework for refreshing hyperdimensional computing," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 53–62.
[14] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cogn. Comput.*, vol. 1, no. 2, pp. 139–159, 2009.
[15] P. Kanerva, "What we mean when we say "what's the dollar of Mexico?": Prototypes and mapping in concept space," in *Proc. AAAI Fall Symp.: Quantum Informat. Cogn. Soc. Semantic Processes*, 2010, pp. 2–6.
[16] P. Kanerva *et al.*, "Random indexing of text samples for latent semantic analysis," in *Proc. Annu. Meeting Cogn. Sci. Soc.*, vol. 1036, 2000.
[17] A. Joshi, J. T. Halseth, and P. Kanerva, "Language geometry using random indexing," in *Proc. Int. Symp. Quantum Interact.*, pp. 265–274, 2016.
[18] O. J. Räsänen and J. P. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 9, pp. 1878–1889, Sep. 2016.
[19] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "VoiceHD: Hyperdimensional computing for efficient speech recognition," in *Proc. IEEE Int. Conf. Rebooting Comput.*, 2017, pp. 1–6.
[20] Y. Kim *et al.*, "Efficient human activity recognition using hyperdimensional computing," in *Proc. 8th Int. Conf. Internet of Things*, 2018, Art. no. 38.
[21] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "HDNA: Energy-efficient DNA sequencing using hyperdimensional computing," in *Proc. IEEE EMBS Int. Conf. Biomed. Health Inform.*, 2018, pp. 271–274.
[22] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "A memory-centric acceleration of clustering using high-dimensional vectors," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2019.
[23] S. Han *et al.*, "Learning both weights and connections for efficient neural network," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
[24] X. Dai, H. Yin, and N. Jha, "NeST: A neural network synthesis tool based on a grow-and-prune paradigm," *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1487–1497, Oct. 2019.
[25] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *IEEE Trans. Comput.*, vol. 68, no. 4, pp. 484–497, Apr. 2019.
[26] M. Imani *et al.*, "SearcHD: A memory-centric hyperdimensional computing with stochastic training," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published, doi: 10.1109/TCAD.2019.2952544.
[27] S. Salamat, B. Khaleghi, M. Imani, and T. Rosing, "Workload-aware opportunistic energy efficiency in multi-FPGA platforms," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2019, pp. 1–8.
[28] B. Khaleghi, S. Salamat, M. Imani, and T. Rosing, "FPGA energy efficiency by leveraging thermal margin," in *Proc. IEEE 37th Int. Conf. Comput. Des.*, 2019, pp. 376–384.
[29] M. Imani, S. Salamat, S. Gupta, J. Huang, and T. Rosing, "FACH: FPGA-based acceleration of hyperdimensional computing by reducing computational complexity," in *Proc. 24th Asia South Pacific Des. Autom. Conf.*, 2019, pp. 493–498.
[30] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proc. Int. Symp. Low Power Electron. Des.*, 2016, pp. 64–69.
[31] M. Imani *et al.*, "Hierarchical hyperdimensional computing for energy efficient classification," in *Proc. 55th Annu. Des. Autom. Conf.*, 2018, Art. no. 108.
[32] M. Imani *et al.*, "QuantHD: A quantization framework for hyperdimensional computing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published, doi: 10.1109/TCAD.2019.2954472.
[33] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2017, pp. 445–456.
[34] A. Rahimi *et al.*, "High-dimensional computing as a nanoscalable paradigm," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 9, pp. 2508–2521, Sep. 2017.
[35] T. F. Wu *et al.*, "Brain-inspired computing exploiting carbon nanotube FETs and resistive RAM: Hyperdimensional computing case study," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 492–494.
[36] H. Li *et al.*, "Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *Proc. IEEE Int. Electron Devices Meeting*, 2016, pp. 16–1.
[37] S. Gupta, M. Imani, and T. Rosing, "FELIX: Fast and energy-efficient logic in memory," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2018, Art. no. 55.
[38] M. Imani, J. Messerly, F. Wu, W. Pi, and T. Rosing, "A binary learning framework for hyperdimensional computing," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2019, pp. 126–131.
[39] S. Datta, R. A. Antonio, A. R. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric IoT," *IEEE Trans. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 3, pp. 439–452, Sep. 2019.
[40] S. Manuel, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 15, 2019, Art. no. 32.
[41] M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, and T. Rosing, "SparseHD: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 53–62.
[42] M. Imani, J. Morris, H. Shu, S. Li, and T. Rosing, "Efficient associative search in brain-inspired hyperdimensional computing," *IEEE Des. Test*, vol. 37, no. 1, pp. 28–35, Feb. 2020.
[43] M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, and T. Rosing, "SparseHD: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *Proc. IEEE 27th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2019, pp. 190–198.
[44] UCI machine learning repository, Accessed: May 2020. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/ISOLET
[45] G. Griffin, A. Holub, and P. Perona, "Caltech-256 object category dataset," California Institute of Technology, 2007.
[46] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[47] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.
[48] M. Everinghametal ., "The pascal visual object classes challenge: A retrospective," *Int. J. Comput. Vis.*, vol. 111, no. 1, pp. 98–136, 2015.
[49] UCI machine learning repository, Accessed: May 2020. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities
[50] A. Reiss *et al.*, "Creating and benchmarking a new dataset for physical activity monitoring," in *Proc. 5th Int. Conf. Pervasive Technol. Related Assistive Environ.*, 2012, Art. no. 40.

Sahand Salamat (Student Member, IEEE) received the BSc degree in electrical and computer engineering from the University of Tehran, Tehran, Iran. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, University of California San Diego, San Diego, California, since 2017. He is a member of System Energy Efficiency Laboratory (SEELab). He is working on accelerating big-data applications (machine learning, database, and bioinformatic) on FPGAs.

Mohsen Imani (Student Member, IEEE) is working toward the PhD degree with the Department of Computer Science and Engineering, UC San Diego, San Diego, California. He is the author of more than 85 publications at top tier conferences and journals. His contributions resulted in several grants funded from multiple governmental agencies (four NSF, three SRC grants) and several companies including IBM, Intel, Micron, and Qualcomm. He has received the most prestigious awards from the UCSD School of Engineering including the Gordon Engineering Leadership Award and the Outstanding Graduate Research Award. He also got several nominations for the best paper awards from multiple conferences. His research interests include brain-inspired computing and computer architecture.

Tajana Rosing (Fellow, IEEE) received the master's degree in engineering management and the PhD degree from Stanford University, Stanford, California, in 2001. She is a professor, a holder of the Fratamico endowed chair, and a director of System Energy Efficiency Lab, UCSD. She is currently heading the effort in SmartCities as a part of DARPA and industry funded TerraSwarm center. During 2009–2012, she led the energy efficient datacenters theme as a part of the MuSyC center. Her research interests include energy efficient computing, embedded and distributed systems. Prior to this, she was a full time researcher with HP Labs while being leading research part-time with Stanford University. Her PhD topic was dynamic management of power consumption. Prior to pursuing the PhD, she worked as a senior design engineer with Altera Corporation.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.