FACH: FPGA-based Acceleration of Hyperdimensional Computing by Reducing Computational Complexity

Mohsen Imani, Sahand Salamat, Saransh Gupta, Jiani Huang, and Tajana Rosing
Computer Science and Engineering, UC San Diego, La Jolla, CA 92093, USA
{moimani,sasalama, sgupta, jih013, tajana}@ucsd.edu

ABSTRACT

Brain-inspired hyperdimensional (HD) computing explores computing with hypervectors for the emulation of cognition as an alternative to computing with numbers. In HD, input symbols are mapped to a hypervector and an assoc iative search is performed for reasoning and classification. An associative memory, which finds the closest match between a set of *learned* hypervectors and a *query* hypervector, uses simple Hamming distance metric for similarity check. However, we observe that, in order to provide acceptable classification accuracy HD needs to store non-binarized model in associative memory and uses costly similarity metrics such as *cosine* to perform a reasoning task. This makes the HD computationally expensive when it is used for realistic classification problems. In this paper, we propose a FPGA-based acceleration of HD (FACH) which significantly improves the computation efficiency by removing majority of multiplications during the reasoning task. FACH identifies representative values in each class hypervector using clustering algorithm. Then, it creates a new HD model with hardware-friendly operations, and accordingly propose an FPGA-based implementation to accelerate such tasks. Our evaluations on several classification problems show that FACH can provide 5.9× energy efficiency improvement and 5.1× speedup as compared to baseline FPGA-based implementation, while ensuring the same quality of classification.

CCS CONCEPTS

• **Computing methodologies → Machine learning approaches**; *Supervised learning*;

KEYWORDS

Brain-inspired computing, Hyperdimensional computing, Machine learning, Energy efficiency

ACM Reference Format:
Mohsen Imani, Sahand Salamat, Saransh Gupta, Jiani Huang, and Tajana Rosing. 2019. FACH: FPGA-based Acceleration of Hyperdimensional Computing by Reducing Computational Complexity. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287624.3287667

1 INTRODUCTION

Machine learning algorithms have shown promising accuracy in many tasks including computer vision, voice recognition, natural language processing, and health care [1–6]. However, the existing machine learning algorithms such as deep neural networks are computationally expensive and require enormous resources to be executed [7–9]. From other hands, embedded devices are often constrained in terms of available processing resources and power budget. Brain-inspired hyperdimensional (HD) computing is a computational paradigm performing energy-efficient cognitive computation. HD works based on the *patterns of neural activity* that are not readily associated with numbers [10]. Due to the very large size of brain's circuits, such neural activity patterns can only be modeled with vectors in high-dimensional space, called hypervectors. HD computing builds upon a well-defined set of operations with random HD vectors and it is extremely robust in the presence of failures. HD offers a complete computational paradigm that is easily applied to learning problems including: analogy-based reasoning [11], latent semantic analysis [12], language recognition [13, 14], prediction from multimodal sensor fusion [15], speech recognition [16, 17], activity recognition [18], DNA sequencing [19], and clustering [20].

HD computing is about manipulating and comparing large patterns, stored in memory as hypervectors. In contrast to existing classification algorithms, such as neural networks, which require significantly complex and costly computation during training and inference [21], HD provides a memory-centric, hardware friendly operations which can be process on light-weight embedded devices. Figure 1 shows the overview of the HD functionality in training and inference phases. In HD training, each input is mapped to a hypervector and then hypervectors for multiple inputs are combined to create class hypervectors. In inference, an associative memory checks the similarity of the input query hypervector with all prestored class hypervectors. For a simple classification task such as language or text classification, HD can use binarized class hypervectors (0 and 1) and simple Hamming distance for similarity check. In this work, we show that in order to achieve acceptable accuracy on realistic classification problems (i.e., speech, activity, or face recognition), HD has to use class hypervectors with non-binary elements, which means that HD needs to use *cosine* metric to find the similarity between query 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. This makes running HD on the general purpose processors slow and energy hungry.

In this paper, we propose a FPGA-based ACceleration of HD (FACH) which significantly reduces the computational cost by removing the majority of the multiplications. FACH employs a clustering algorithm in order to share the values in each class hypervector by taking into account the statistical properties of each operand and output within the HD class. Instead of multiplying all pairs of the query and a class hypervector, FACH 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. This significantly accelerates HD computation by reducing the number of required multiplications. Based on this technique, we

ASPDAC '19, January 21–24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3287667

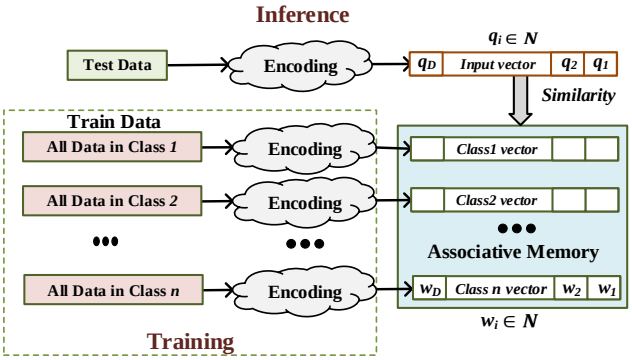


Figure 1: HD functionality in train and inference phases using encoding and associative memory modules.

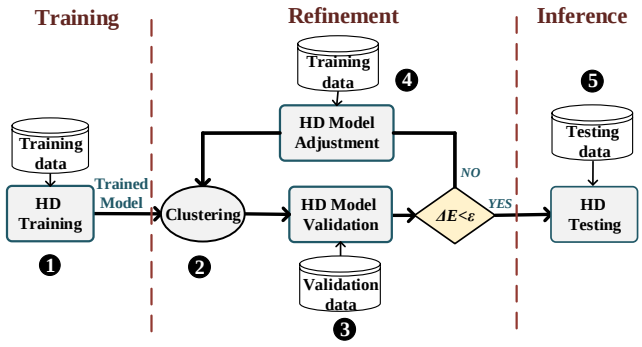


Figure 2: (a) FACH supporting Framework consisting of HD training to create initial class hypervectors.

create a new HD model with hardware-friendly operations, and accordingly propose an FPGA-based acceleration for such tasks. Our evaluations on several classification problems show that FACH can provide 5.9× energy efficiency improvement and 5.1× speedup as compared to baseline FPGA-based implementation while ensuring the same quality of classification.

2 HD COMPUTING ALGORITHM

2.1 FACH Overview

HD provides a general model of computing which can apply to different type of learning problems. Classification is one of the most important supervised learning algorithm. To perform classification, HD uses two main modules: encoding and associative memory. Figure 1 shows the overall structure of the HD classification in both training and inference phases. Encoding module maps input data to a vector in high dimensional space, called hypervector. Training performs on hypervectors, by adding all hypervectors corresponding to a particular class together. In a similar way, HD creates a single hypervector for each existing class. These class hypervectors store in an associative memory. In inference, HD uses the same encoding scheme to map a query data to high-dimensional space. Finally, the associative memory performs the reasoning task by looking for a class hypervector which has the highest similarity to input hypervector. In the following, we briefly explain how each module works.

2.2 Encoding Module

The first step in HD computing is to encode input data to high-dimensional vectors. The main goal of encoding module is to map input data to hypervector with D dimensions (e.g. $D = 10,000$), while keeping all information of a data point in the original space, e.g., the feature values and their indexes for feature vector. Input data can have different representations, thus there are different encoding modules to map data to high dimensional space. For example, work in [16, 22] proposed encoding methods to map feature vectors to high dimensional space. Work in [23] encodes text-like data using the idea of random indexing.

2.3 HD Model Training

HD performs the training procedure on the encoded hypervectors. For all data corresponding to a particular class, HD 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 a class i^{th}. The HD model can be generated by adding all hypervectors with the same tag as $\mathbf{C}^i = \sum_j Q_j^i$, where $\mathbf{C}^i = \{w_1^i, w_2^i, \ldots, w_D^i\}$. Each element of the class hypervector can have non-binarized value ($w \in N^D$). This significantly increases the inference cost, as the rest of reasoning task, i.e., similarity check, needs to perform using integer rather than binary values.

To reduce the computational cost, several prior works tried to binarize the class elements after training by applying a *majority* function on each dimension [23, 24]. However, this reduces the amount of information stored in each class hypervector. Later in this section, we discuss the accuracy-efficiency trade off when using binarized or non-binarzied class hypervectors for classification.

2.4 Associative Memory Module

After training, all class hypervectors are stored in an associative memory (shown in Figure 1). In inference, an input data encodes a *query hypervector* using the same encoding module used for training. The associative memory is responsible for comparing the similarity of the input query hypervector with all stored class hypervectors and selecting a class with the highest similarity. For all classification problems, HD uses the same associative search to perform the reasoning task, regardless of the encoding module. Associative memory 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 a inexpensive and suitable similarity metric, while class hypervectors with non-binarized elements need to use *cosine* for similarity check. Most existing HD computing techniques are using binarized class hypervectors in order to eliminate the costly *cosine* metric [17, 24]. However, we observed that HD with binary model provides significantly low classification accuracy as compared to non-binary model. For example, for face recognition, HD using non-binarized class elements provides 57.8% higher accuracy than HD using binarized hypervectors. From other hand, HD with non-binary model involves large amount of multiplications. For example, an application with k class hypervector and D dimensions involve $k \times D$ multiplications.

3 FACH FRAMEWORK

3.1 Overview

In this section, we propose a FPGA-based acceleration of HD (FACH), which exploits the statistical characteristic of the hyperdimensional computing in order to reduce the HD computational complexity. Figure 2a shows the overview of the FACH framework consisting of three main steps: training, model refinement, and inference. As we explained in Section 2.3, HD encodes all data points to hypervectors

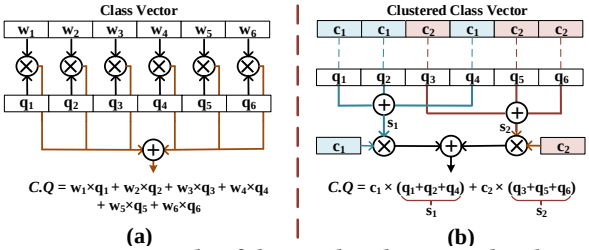


Figure 3: An example of dot product between the class and query vectors with six dimensions (a) using conventional method, (b) when the elements of the class vector clustered.

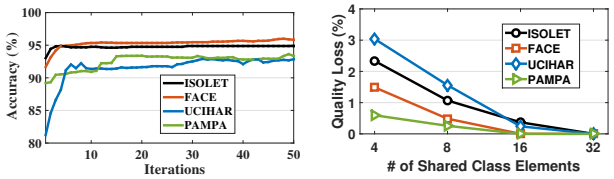


Figure 4: a) The classification accuracy of applications during retraining iterations. b) Impact of number of class elements on the quality loss of different applications.

and trains the HD model by combining data points corresponding to each class (❶). HD refinement clusters the values that elements in each class hypervector can take by applying non-linear clustering on the trained class hypervectors (❷). This method reduces the possible values that the elements of each class hypervector can take. Next, FACH estimates the accuracy of the new HD model on the validation data (validation is part of training data). If the error rate is larger than a pre-defined ϵ value, FACH 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 (❸). When the convergence condition satisfied, FACH framework sends a new HD model with the clustered class elements to inference in order to perform the rest of classification task (❹). Finally, FACH uses the modified HD model with clustered class elements for inference (❺). In this following, we explain the details of the FACH framework functionality.

3.2 Reduction in Multiplication Domain

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 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 i^{th} 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 FACH reduces the class span by carefully selecting a subset from the input spaces, called "best representatives". FACH 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 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 addition is multiplied by the value of that particular class.

Here we explain how FACH 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 on the the pre-trained class hypervectors. For each class hypervector, our design identifies a specified number of clusters, say k, based on clustering algorithm. 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 ($c \in N$).

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

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

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

where θ_i is the i^{th} 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.

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

This method reduces the number of multiplications involved in dot product from D to k, where k can be about three order of magnitudes smaller than D. Figure 3 shows an example of the dot product between a class and a query vector using conventional method and clustered model. Since in conventional method the class elements can take any value, the dot product involve six multiplications (Figure 3a). FACH exploits the advantage of clustered class values in order to first add the query elements corresponding to the same centroid and then multiply the result with the centroid values (Figure 3b). 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 $\Delta E < \epsilon$, FACH 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

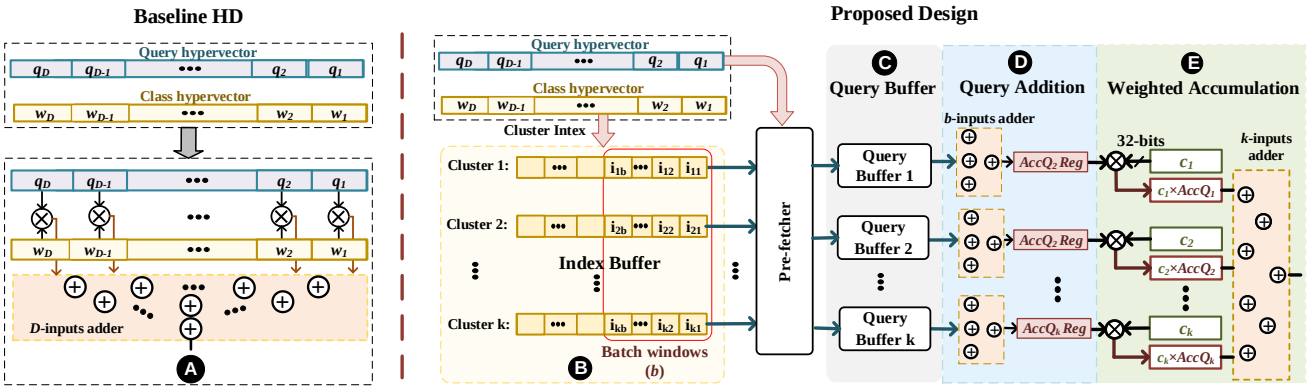


Figure 5: The hardware support to measure dot product between a query and class hypervectors in baseline HD and FACH with clustered class elements.

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 i^{th} class hypervector (C^i) while it actually belongs to j^{th} class (C^j), our retraining procedure subtracts the input hypervector from the i^{th} class and add it to j^{th} 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. 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.

Figure 4a 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 the quality loss due to clustering by using less than 50 iterations. All pre-processing operations in the HD refinement module are performed offline and their overhead is amortized among all future executions of FACH accelerator. Figure 4b shows the final quality loss, ΔE, when FACH clusters the class hypervector to different 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 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 FACH using different clusters.

4 HD HARDWARE ACCELERATION

In this work, we implement baseline HD and FACH on a FPGA. In the following, we explain how each design can be accelerated on FPGA.

4.1 Baseline HD Acceleration

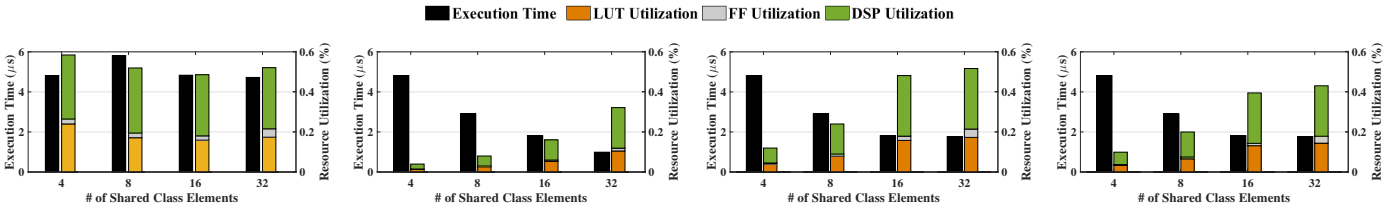
We use FPGA to accelerate HD computing inference. Figure 5*A* shows that the FPGA-based implementation of the baseline HD requires D parallel multiplications to calculate the dot product between the query and class hypervectors. Then, the results of all D multiplications accumulate in a tree-based adder. However, when D is large, FPGA does not have enough resources to perform multiplications in all dimensions in parallel (Ⓐ). The number of input dimensions which FPGA reads at a time depend on the number of classes, and the number of available Digital Signal Processors (DSPs) in FPGA. We implement HD on the Kintex-7 FPGA KC705 Evaluation Kit with 840 DSPs. In this case, our design sequentially reads the first d elements of the query vector and multiply it to corresponding class elements ($d < D$). Then, the computation on the rest of query elements are performed sequentially.

4.2 FACH Acceleration

Figure 5 illustrates the FACH 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 FACH, 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 Figure 5B, stores all class indices which have the value as c_1. Since each class has D dimensions, we require log_2**D** bits to store each index.

Due to resource limitation in FPGA, we can only read d 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 d elements in a read window might belong to any of the clusters. In this case, each index buffer requires a tree-based adder with d inputs in order to take care of the worse case scenario, when all d query dimensions correspond to a single cluster. Instead, in this work, each read window accesses to $b = d/k$ 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 speedup the computation, FACH stores the index buffers, which are actually 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 windows, FACH pre-fetches the query elements and store them in query buffers (Ⓒ). Next to each query buffer, a tree-based adder accumulates all



(a) ISOLET (b) FACE (c) UCIHAR (d) PAMPA

Figure 6: Execution time and average resource utilization of FPGA running FACH with different number of shared elements.

d/k indices corresponding to a particular centroid (❸). The results of these additions are stored in registers. Next, FPGA processes the next batch sequentially. FACH 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 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 FACH 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 (❹). 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 FACH has been implemented with software and hardware modules. For software support, we exploit Scikit-learn library [25] for clustering and C++ software implementation for the HD model training and verification. For hardware support, we use FPGA to accelerate HD computation. We fully implemented FACH inference functionality using Verilog. We verified the functionality of the design using both synthesis and real implementation of the FACH on Xilinx Vivado Design Suite [26]. The synthesis code was implemented on the Kintex-7 FPGA KC705 Evaluation Kit.

5.2 Workloads

We evaluate the efficiency of the proposed FACH 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 [27]. **Face Recognition (FACE):** We exploit Caltech dataset of 10,000 web faces [28]. Negative training images, i.e., non-face images, are selected from CIFAR-100 and Pascal VOS 2012 datasets [29]. **Activity Recognition (UCIHAR)** [30]: 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 (PAMPA) [31]: 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 was corresponded to an activity ID. The goal is to recognize 12 different activities.

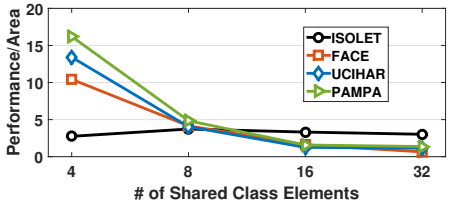


Figure 7: Performance per utilization of FPGA running applications with different number of shared class elements.

5.3 Accuracy-Efficiency Trade-off

Figure 6 shows the execution time and average resource utilization of FPGA while running four applications. The resource utilization shows the average utilization of LUT, FF and DSP in the FPGA. The x-axis shows the number of shared elements (centroids) in each class hypervector. Comparing the results of baseline HD with the FACH show that FACH can significantly improve the efficiency of the HD computing by reducing the number of multiplications. FACH performance depends on the number of shared class elements. FACH with more number of centroids requires more FPGA resources and thus consumes higher power. However, it improves the performance of FACH by increasing the parallelism. For example, increasing the number of centroids from 4 to 32 improves the FACH performance by 2.12× while utilizes on average 2.08× more resources.FPGA performance does not improve linearly for a model with larger than 16 shared class elements. This is because larger FACH models, for example a model with 32 shared elements, does not fit on FPGA, therefore FPGA processes the FACH sequentially.

As we discussed in section 3.1, FACH accuracy depends on the number of shared class elements. The more is the number of shared elements, the higher the accuracy FACH can provide. Our evaluations show that FACH on average can achieve 5.9× better energy efficiency and 5.1× faster execution as compared to the baseline FPGA-based HD implementation while providing the same quality of classification. Similarly, accepting less than 1% quality loss, FACH can provide 6.5× energy efficiency improvement and 4.9× speedup as compared to baseline FPGA-based implementation of HD.

Figure 7 shows the performance per average utilization for FPGA while running applications with different number of clustered centroids. Using this metric, we observe that although FACH with a larger number of cluster centroids has higher performance, performance per resource utilization is higher for FACH using less number of centroids. In other words, the FPGA can better utilize the resources while running FACH with a smaller number of shared

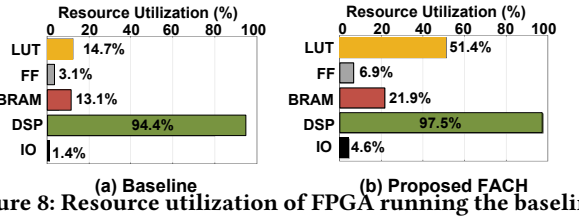


Figure 8: Resource utilization of FPGA running the baseline HD and proposed FACH with 8 shared class elements.

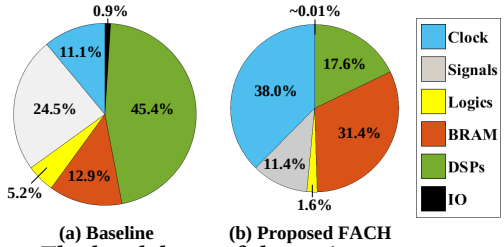


Figure 9: The breakdown of dynamic power consumption of the FPGA implementing baseline HD and FACH with 8 shared class elements.

class elements. Thus, we can maximize the FPGA efficiency by using the minimum number of cluster centroids which provide the acceptable quality of loss.

5.4 Utilization/Power Breakdown

Figure 8 shows the breakdown of the FPGA resources while implementing the baseline HD and FACH for UCIHAR with 8 cluster centroids. For the baseline HD, FPGA uses several multiplications to measure the similarity of a query and class hypervector. This increases the DSP utilization to 94.4% while LUT and BRAM have only 14.7% and 13.1% utilization. In fact, in this implementation the computation performance is limited by the number of DSPs available on the chip. In contrast, FACH with shared class elements can better utilize the FPGA resources by significantly reducing the number of required multiplications. FACH implementation uses DSPs in order to add the query elements which have been pre-fetched to query buffer. Although this increases the BRAM utilization, it allows FPGA to access the query values at a much faster rate in order to fully utilize the DSPs. In addition, FACH exploits the distributed memories, designed by LUT and FF in order to store the index buffer. This increases the utilization of LUT and FF to 51.4% and 6.9% respectively.

Figure 9 also shows the power breakdown of FPGA while implementing baseline HD and proposed FACH. The results show that for baseline HD implementation, DSP takes 45.4% of total power consumption, while BRAM and logic together take around 18.1% of the power. In contrast, FACH implementation requires higher BRAM utilization which increases the contribution of BRAM to total power to 31.4%. Moreover, in FACH implementation, clock takes 38.0% of total power, mostly to implement distributed memory to store index buffer and perform pre-fetching.

6 CONCLUSION

We propose a novel hyperdimensional computing framework, called FACH, which significantly reduces the cost of classification. The framework extracts representative operands of a trained HD model using clustering algorithm. 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 FACH can provide 5.1× faster execution and 5.9× higher energy efficiency as compared to the baseline HD.

ACKNOWLEDGEMENTS

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

REFERENCES

[1] K. Kourou *et al.*, "Machine learning applications in cancer prognosis and prediction," *Computational and structural biotechnology journal*, vol. 13, pp. 8–17, 2015.
[2] F. Imani *et al.*, "Nested gaussian process modeling for high-dimensional data imputation in healthcare systems," in *IISE 2018 Conference & Expo, Orlando, FL, May*, pp. 19–22, 2018.
[3] M. Libbrecht *et al.*, "Machine learning applications in genetics and genomics,"
[4] M. Imani *et al.*, "MFBO-SSM: Multi-fidelity Bayesian optimization for fast inference in state-space models," in *AAAI*, 2019.
[5] Y. Kim *et al.*, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *ICCAD*, pp. 25–32, IEEE, 2017.
[6] M. Imani *et al.*, "Bayesian control of large mdps with unknown dynamics in data-poor environments," in *Advances in Neural Information Processing Systems*, 2018.
[7] M. Imani *et al.*, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.
[8] Y. Kim *et al.*, "Image recognition accelerator design using in-memory processing," *IEEE Micro*, 2018.
[9] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *DATE*, pp. 1775–1780, IEEE, 2017.
[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, "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*, pp. 2–6, 2010.
[12] P. Kanerva *et al.*, "Random indexing of text samples for latent semantic analysis," in *CogSci*, vol. 1036, Citeseer, 2000.
[13] A. Joshi *et al.*, "Language geometry using random indexing," *Quantum Interaction 2016 Conference Proceedings*, In press.
[14] M. Imani *et al.*, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.
[15] O. R. andothers, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. PP, no. 99, pp. 1–12, 2015.
[16] M. Imani *et al.*, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *ICRC*, pp. 1–6, IEEE, 2017.
[17] M. Imani *et al.*, "Hierarchical hyperdimensional computing for energy efficient classification," in *Proceedings of the 55th Annual Design Automation Conference*, p. 108, ACM, 2018.
[18] Y. Kim *et al.*, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, p. 38, ACM, 2018.
[19] M. Imani *et al.*, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *Biomedical & Health Informatics (BHI), 2018 IEEE EMBS International Conference on*, pp. 271–274, IEEE, 2018.
[20] M. Imani *et al.*, "A memory-centric acceleration of clustering using high-dimensional vectors," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019.
[21] S. Han *et al.*, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, pp. 1135–1143, 2015.
[22] M. Imani *et al.*, "A binary learning framework for hyperdimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019.
[23] A. Rahimi *et al.*, "High-dimensional computing as a nanoscalable paradigm," *TCAS I*, 2017.
[24] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *HPCA*, pp. 445–456, IEEE, 2017.
[25] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *JMLR*, vol. 12, pp. 2825–2830, 2011.
[26] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.
[27] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.
[28] G. Griffin *et al.*, "Caltech-256 object category dataset," 2007.
[29] M. Everingham *et al.*, "The pascal visual object classes challenge: A retrospective," *IJCV*, vol. 111, no. 1, pp. 98–136, 2015.
[30] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities.
[31] A. Reiss *et al.*, "Creating and benchmarking a new dataset for physical activity monitoring," in *PETRA*, p. 40, ACM, 2012.