2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00082

Full Stack Parallel Online Hyperdimensional Regression on FPGA

Hanning Chen*, M.Hassan Najafi†, Elaheh Sadredini‡ and Mohsen Imani*
*University of California, Irvine, Irvine, CA 92697, USA
†University of Louisiana at Lafayette, Lafayette, LA 70503, USA
‡University of California, Riverside, Riverside, CA 92521, USA
Email: {hanningc, m.imani}@uci.edu, najafi@louisiana.edu, elaheh@cs.ucr.edu

***Abstract*—Hyperdimensional computing (HDC) has been proposed to more closely model the brain from the abstract and functionality level. Compared to the traditional sequential regression model, HDC based regression model naturally supports parallel operation, making it an ideal algorithm to be accelerated on the FPGA platform. In this paper, we propose HyDRAF, an FPGA acceleration of hyperdimensional regression supporting online learning. To overcome the computation overhead from the long-size hypervector, we introduce multiple FPGA optimizations to efficiently handle long vector access, such as on-chip storage partitioning. Furthermore, we optimize the model update process by using efficient sparse matrix representation. We also integrate the encoding module into the accelerator to realize online training by reducing off-chip DRAM access, thus enhancing FPGA resource utilization. We also evaluate the effectiveness of our approach on a wide range of regression problems. Our results show that the FPGA platform provides, on average, 11.8× speedup and 27.5× energy efficiency compared to the state-of-the-art regression method running on NVIDIA GTX 1080 GPU. On a Xilinx Alveo U200 accelerator card platform drawing less than 4 Watt for kernel Virtex Ultrascale+ XCU200 FPGA, HyDRAF demonstrates up to 1.2 million data classifications per second.**

I. Introduction

Regression is supervised learning which is used to predict continuous values. It is widely used to estimate the relationship between a dependent variable. Regression is applied to predict the outputs, forecast the data, analyze the time series, and find the causal effect dependencies between the variables [1]. Regression techniques need to rely on sophisticated and costly deep learning algorithms. However, running these algorithms during training results in significant computational power and storage, which is beyond the capability of existing edge devices [2]. As a result, many devices cannot enable on-device learning; thus, they stream most data to the cloud for analysis. This data transmission leads to scalability, security, and privacy concerns. Therefore, it is essential to enable robust, scalable, and real-time learning on embedded devices with limited computing capability and off-chip memory.

Hyper-Dimensional Computing (HDC) is introduced as an alternative computing model mimicking crucial brain properties [3], [4] for energy efficiency and robust computation. HDC is motivated by the observation that the human brain operates on high-dimensional data representations. In HDC, objects are thereby encoded with high-dimensional vectors, called *hypervectors*, which have thousands of elements [5]. HDC incorporates learning capability along with typical memory functions of storing/loading information. It mimics important functionalities of the human memory model with vector operations, which are computationally tractable and mathematically rigorous in describing human cognition. HDC provides several advantages as compared to existing deep learning solutions: (1) being highly parallel and suitable for online on-device learning [5], (2) exposing hidden features; enabling single-pass learning with just a few samples [6], [7], and (3) being robust against noise and corrupted data [8], [9].

Since the HDC needs to compute many values for a single operation, the conventional CPU-centric architecture would not be the best platform to run the HDC applications. HDC is easily parallelizable and can benefit from hardware accelerators. Prior work showed how the high-dimensional and parallel nature of HDC is ideal for acceleration in traditional hardware platforms, such as FPGA and ASIC [10], [11], [12], [13]. However, prior work primarily focused on HDC classification acceleration [11], [12], [14], [15]. In contrast, in this paper, we present a design to accelerate hyperdimensional regression and support online learning on various different FPGA platforms. Our solution, called HyDRAF (Hyperdimensional Regression Accelerator on FPGA), introduces several architectural optimizations to maximize throughput by getting the best use of FPGA resource utilization. Here are the main contributions of the paper:

- HyDRAF is an online learning framework for accelerating hyperdimensional regression on FPGA. Our regression exploits hyperdimensional primitives to encode raw data into high-dimensional space. Then, it performs the model learning process by similarity checking of the distance of an encoded query with a model in high-dimensional. By implementing HD encoding, training, and inferring at the same platform, our solution, therefore, is a real full-stack HD computing accelerator.
- We conduct hardware/software co-design targeting online regression task. First, we introduce an on-chip storage partitioning to efficiently handle long vector. Second, we optimize the model update process by using efficient sparse matrix representation. We integrate the encoding module into the accelerator to realize online training by reducing off-chip DRAM access (I/O utilization), thus enhancing FPGA resource utilization.
- Unlike the existing method that encodes data once offline, our solution explores the opportunity of iterative data encoding. Our solution stores raw data in off-chip DRAM and re-

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00082

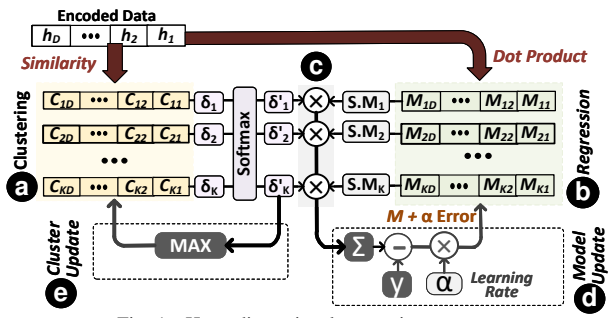

Fig. 1. Hyperdimensional regression process.

encode data in every iteration. On the one hand, this ensures the off-chip memory bandwidth is not a computational bottleneck. On the other hand, this reduces on-chip data storing overhead. Encoding data on-chip also enhances the training and inferring process's flowing capabilities.

We also evaluate the effectiveness of our approach on a wide range of regression problems. Our results show that the FPGA platform provides, on average, 11.8× speedup and 27.5× energy efficiency compared to the state-of-the-art regression methods running on NVIDIA GTX 1080 GPU. On a Xilinx Alveo U200 accelerator card platform drawing less than 4 Watt for kernel Virtex Ultrascale+ XCU200 FPGA and total 41.7 Watt for the whole board, HyDRAF demonstrates up to 1.2 million data classifications per second. During training, HyDRAF also processes 0.32 million data per second at each epoch, enabling online learning from the data stream.

II. Hyperdimensional Regression

Hyperdimensional Computing (HDC) is introduced by neuroscientists as an alternative computing method to model human memory [3], [12]. HDC mimics crucial properties of human memory using high-dimensional vectors, called *hypervectors*. For example, the brain efficiently aggregates and memorizes the relationship between data. In the HDC, the addition of hypervectors imitates the data aggregation, and we can quantify the inter-data relationship based on the hypervector similarity.

Figure 1 shows an overview of Hyperdimensional regression [4]. The first step in HDC is to map each data points into high-dimensional space. The mapping procedure is often referred to as *encoding*. The regression function operates over encoded data. During regression, we first create two sets of models: *Cluster Model* to cluster data points with high similarity, and *Regression Model* to perform the prediction. Each model consists of multiple vectors with the same dimensionality as encoded data points. Each vector in the regression model corresponds to a cluster of inputs aggregated in a cluster hypervector. During training, HyDRAF first checks the similarity of a data point with the input model. Depending on the search result, we update the cluster and regression model accordingly. Here, we explain the details of HDC regression.

Hyperdimensional Encoding: Encoding is the first operation involved in HDC. Here, we consider the state-of-the-art encoding method for feature vector. Let us consider an encoding function that maps a feature vector $\vec{F} = \{f_1,\ f_2, \ldots,\ f_n\}$, with n features ($f_i \in \mathbb{R}$) to a hypervector $\vec{\mathcal{H}} = \{h_1,\ h_2, \ldots,\ h_D\}$ with D dimensions ($h_i \in \mathbb{R}$). We generate each dimension of encoded data by calculating: $\vec{\mathcal{H}} = \sum_{k=0}^{n-1} f_i \cdot \vec{\mathcal{B}}_i$, where $\vec{\mathcal{B}}_i \in \{0,1\}^D$ are randomly generated base hypervectors. Since randomly generated hypervectors are nearly orthogonal ($\delta(\vec{\mathcal{B}}_{i_1}, \vec{\mathcal{B}}_{i_2}) \simeq 0$, where δ denotes the cosine similarity), each base hypervector can retain the spatial or temporal location of each feature in an input.

Model Learning: Let us assume HyDRAF with k models. HyDRAF stores two sets: cluster hypervectors ($\mathbf{C} = \{\vec{\mathcal{C}}_1, \vec{\mathcal{C}}_2, \cdots, \vec{\mathcal{C}}_K\}$) and model hypervectors ($\mathbf{M} = \{\vec{\mathcal{M}}_1, \vec{\mathcal{M}}_2, \cdots, \vec{\mathcal{M}}_K\}$). The cluster hypervectors are initialized to random binary values, while model hypervectors are initialized as zero hypervectors. Figure 1 shows the functionality of regression over encoded data. We first check the similarity of $\vec{S}$ with all cluster hypervectors. Each similarity value shows the confidence that a data point belongs to that cluster (ⓐ). For the same encoded data, we also perform regression on the k available models (ⓑ). Then, we predict the output value using all models and their corresponding confidence value (ⓒ): $\hat{y} = \sum_{i=1}^{K} \delta(\vec{S}, \vec{\mathcal{C}}_i)\ \vec{\mathcal{M}}_i.\vec{S}$. The predicted value is the weighted accumulation of all regression models. The weight of each model, $\delta(S, C_i)$, determines the confidence of each cluster center for having $\vec{S}$. During training, HyDRAF updates the model based on how far is this prediction from the actual output value (ⓓ): $\vec{\mathcal{M}}_i \leftarrow \vec{\mathcal{M}}_i + \alpha(y - \hat{y}) \times \vec{S}$, where term '$y - \hat{y}$' indicates the error between the actual output and predicted result and 'α' a hyperparameter that controls the speed of model update during the training phase. Our regression model continues iterative updates over training data points until the quality of regression stabilizes during the last few iterations.

III. HyDRAF Overview

The hyperdimensional regression model has sequential computing process [4]. The encoding module, clustering/regression, and model update are happening one after each other to perform a regression task. This sequential computing is suitable for CPU-centric architecture as CPUs have access to limited resources. CPU is more suitable for control logic complex computation but less parallel computation. However, accelerators often have a huge number of resources that could use to parallelize the regression process. This makes it more suitable to be realized on computing devices with a large volume of computation units, such as FPGA and GPU. Unlike neural network, HDC operates over low-precision values that makes it optimal for acceleration on FPGA platform [12]. However, the long size of the hypervector makes it hard to naively move HDC design into FPGA. For fully parallel HDC computation and high throughput regression, it is necessary to fully utilize on-chip resources. Previous work [4] only proposed using HDC for regression task, but on the one hand didn't optimized its regression algorithm based on FPGA's available resources, on the other hand is a sequential learning process. In this section, we conduct a hardware/software co-design and introduce multiple optimization techniques to accelerate HDC regression on a wide range of FPGA platforms.

A. Matrix-Based HD regression

HDC regression is performed originally sequentially for each training sample [4]. However, to utilize FPGA parallel computing capability, we develop a solution to train regression on a batch of data. The batch-based training maximizes the FPGA resources utilization and also balance FPGA I/O overhead. In FPGA, I/O utilization (i.e., AXI4 bandwidth) significantly impacts the required time to load data from off-chip memory (e.g., DRAM) to FPGA. Here, we propose a matrix-based, parallel HDC regression model enabling highly parallel batch training. Our solution significant speeds up the training computation.

B. Model Update

Let us assume our regression model has K cluster hypervectors ($\mathbf{C}$) and accordingly k regression hypervector ($\mathbf{M}$). We suppose the size of each hypervector is D. The purpose of using cluster hypervector is to select regression hypervector according to each input hypervector. The cluster hypervector matrix is initiated to random values, while the regression hypervector matrix is initiated with all 0 elements. Let us assume a batch of encoded data with size of N: $\mathbf{H} = \{\vec{H}_1, \vec{H}_2, ..., \vec{H}_N\}$.

To update each features of our regression model, the first step is to generate the confidence matrix $\mathbf{\Delta}$. The calculation is conducting inner production of training batch matrix $\mathbf{H}$ and cluster matrix $\mathbf{C}$: $\mathbf{\Delta} = \mathbf{H} \cdot \mathbf{C}^T$, where $\mathbf{C}^T$ indicates a transposed clustering matrix. The generated confidence matrix, $\mathbf{\Delta}$, has a size of $K \times N$. In this matrix, each row corresponds to one training data in a batch, and the column represents the number of clusters. For example, $\mathbf{\Delta}_{ij}$ represents i^{th} train data in a batch and its corresponding confidence to cluster j^{th}. We also apply *softmax* function to normalize $\mathbf{\Delta}$ to get $\mathbf{\Delta}'$. The final confidence matrix $\mathbf{\Delta}'$ is like a weight matrix in a neural network which will be used later. Parallel with the clustering process; we also predict the regression outcome using input $\mathbf{H}$ and regression matrix $\mathbf{M}$: $\mathbf{P} = \mathbf{H} \cdot \mathbf{M}^T$.

To compute the regression result, our solution weights every regression model using the confidence matrix $\mathbf{\Delta}'$ obtained from cluster model. We call the weighted prediction result, $\mathbf{Y} = \mathbf{P} \otimes \mathbf{\Delta}'$. The reduction of this outer product on cluster direction gives us $\vec{Y}_r$. Here, Y_{r_i} is the prediction of the i^{th} training data in the batch.

Model Update We use the prediction result to update the cluster hypervector matrix $\mathbf{C}$ and regression hypervector matrix $\mathbf{M}$. We will calculate loss first for each training batch and use the loss with the training batch to update the model. The loss calculation is: $\vec{E} = \vec{Y}_r - \vec{Y}_t$, where $\vec{Y}_t$ is the label of the training batch. For regression hypervector matrix update, each member of training batch will be multiplied with its prediction loss and added to each clusters of regression matrix:

$$\mathbf{M}^i_{new} = \mathbf{M}^i_{old} + \alpha * \sum_{j=1}^{N} \mathbf{E}_j * \mathbf{H}_j \quad \text{for} \quad i \in [1, K]$$

Cluster Update: For cluster hypervector matrix, the update process is more complicate. Since each hypervector is able to store limited information, we only need to update each training batch's member for a cluster hypervector that has the highest confidence. Otherwise, our regression results will be overfitted. We use a select matrix, called $\mathbf{S}$, which has $N \times K$ size. The rows and columns of the select matrix represent the cluster and batch indices, respectively. The value of matrix $\mathbf{S}$ is shown below:

$$\mathbf{S}_{i,j} = \begin{cases} 1 & \max\{\mathbf{\Delta}_j\} == \mathbf{\Delta}_{i,j} \\ 0 & \text{otherwise} \end{cases}$$

This indicates that for every element of select matrix $\mathbf{S}$, for example, $S_{i,j}$ will be 1 only when the training batch's jth data's highest confidence index is cluster i. Otherwise, the value of $\mathbf{S}$ is 0. After having the select matrix, we can update the cluster hypervector matrix:

$$\mathbf{C}_{new} = \mathbf{C}_{old} + \mathbf{S} \cdot \{\mathbf{L}^T \cdot \mathbf{H}\} \quad (1)$$

In other words, we first compute the dot product of $\mathbf{L}^T$ and $\mathbf{H}$, so each encoding hypervector of the training batch is multiplied with its corresponding loss value. We call the production result as $\mathbf{H}_{new}$. Then, we calculate the dot product of the select matrix with this updated training batch hypervector matrix $\mathbf{H}_{new}$. In this way, all training hypervector that should update the same cluster hypervector will be accumulated.

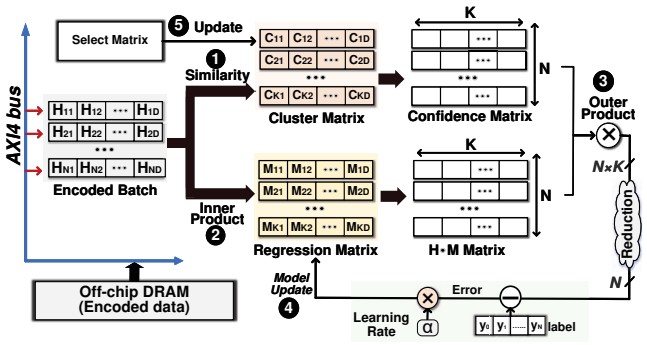

Fig. 2. Top FPGA Microarchitecture Design.

C. Prediction

After training the regression model, we can use the model to predict new data. There are two steps for new data prediction. The first is to use the same encoding method as mentioned in the training process to map the new data into high-dimension. Here, we only need to encode one data point instead of a batch of data. The second step is to compute the similarity of encoded hypervector with the cluster matrix $\mathbf{C}$ to generate confidence vector $\mathbf{\Delta}_p$. Finally, we perform the prediction using the following equation: $Y = \sum_{i=1}^{K} \mathbf{\Delta}_{pi}(\mathbf{M}_i \cdot \mathbf{H})$.

IV. HyDRAF Optimization

Our solution performs regression over a batch of encoded data coming from off-chip DRAM via AXI Interconnect, based on AMBA AXI4 bus protocols. In the first step, HyDRAF computes the similarity of a query with cluster hypervectors (❶). Meanwhile, HyDRAF also predicts the regression outcome between model matrix and query (❷). With batch query hypervector input, both two operations (❶ and ❷) are matrix-matrix multiplication (M2MM). The regression

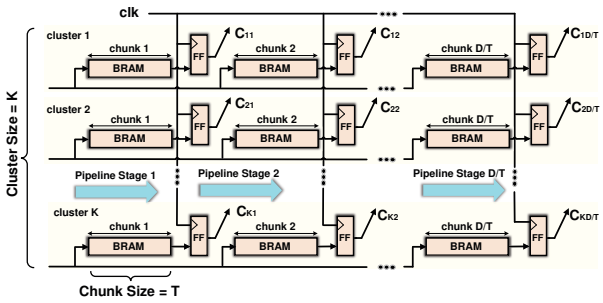

Fig. 3. Hypervector Fragmentation & Storage Partitioning.

prediction is weighted by confidence matrix (❸), which is a two-matrix outer product operation. After calculating the loss result, we update the regression model and cluster model (❹ and ❺). Updating cluster model is much harder since in naive implementation, to avoid memory access conflict, there is two M2MM operations during the updating process (❺). In this section, we analysis the main challenges of implementing our HDC-based regression on FPGA. Then, we present our optimization techniques for highly optimized and efficient regression implementation on FPGA.

A. Hypervector Fragmentation

Unlike traditional neural networks, HDC regression operates over hypervectors using well-defined hyperdimensional primitives. HDC works transparently based on information theory. The capacity of each hypervector to memorize or learn information can be mathematically defined. This capacity depends on two factors: (1) dimensionality of a hypervector, (2) precision of each hypervector element. In HDC, there is a trade-off in selecting dimensionality and precision. Using low precision hypervectors (e.g., single-bit), HDC needs to rely on very long hypervectors for computation. On the other hand, by increasing the precision of each element, HDC can operate over shorted size vectors. This shorter size should still be long enough to ensure nearly orthogonal representation. For example, a binary hypervectors with $D = 10k$ has a similar theoretical capacity as an 8-bit precision hypervector with $D = 1k$. The selection between high or low precision hypervectors depends on the underlying hardware. For example, FPGA has access to several low-cost lookup tables (LUTs).

We partition the long hypervector into several sizes of T chunks where each chunk consists of an equal number of dimensions with a certain precision (e.g., 4-bits). This will fragment a single hypervector into $\frac{D}{T}$ chunks. These chunks will be pre-fetched into on-chip memory (shown in Figure 3). In our architecture, the complexity of operations relates to the precision of hypervector elements, and the dimensionality of matrix multiplication relates to chunk size (T).

B. On-chip Storage Partitioning

We partition all data stored on-chip, including cluster model matrix, regression model matrix, and training data matrix. This partitioning parallelizes the matrix multiplication and pipelines the computation. During the FPGA design process, we partition the original 2-dimensional matrix into 3-dimensional. Figure 3 shows the cluster matrix stored in

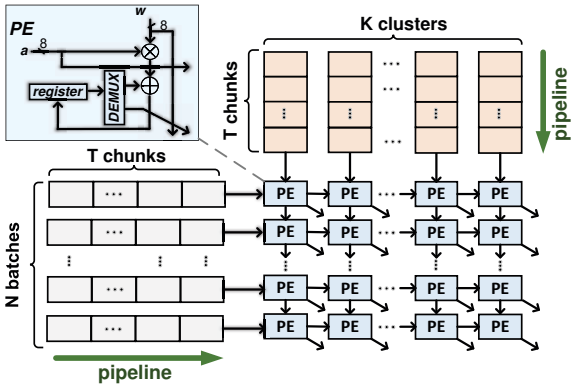

Fig. 4. M2MM accelerator microarchitecture design.

partitioned BRAM blocks. The first dimension is the index of the chunks (Figure 3). The matrix operation size depends on the chunk size (T); increasing a separate dimension here will make the later accelerator's pipeline design much more manageable. In this way, later M2MM operation and on-chip BRAM update will be chunk by chunk pipeline. The second dimension is the number of clusters or the batch size (Figure 3). For each cluster, M2MM will be independent read and write to speed up the execution. The third dimension represents the size of each chunk. In our design, the first and second dimensions of on-chip BRAM are fully partitioned to pipeline the matrix multiplication through chunks and parallelize the multiplication and addition inside chunks.

C. Systolic Array Acceleration

Prior HDC researches exploit vector-vector or vector-matrix operations, which mainly use tree adders and single boolean logic to finish the computation. On FPGA with massive parallel operation supported, it is necessary to use M2MM operation. In this work, we leverage systolic array to enable parallel hypervector multiplication. As shown in Figure 4, we take similarity check between query and cluster matrix as example to illustrate the design. The height of the systolic array is the batch size, N, and its width is cluster size, K. There will be $N \times K$ processing elements (PE) in the whole array. Each PE consists of a 8-bits to 8-bits (depending on the data type of the accelerator, it could be 4-bits to 4-bits) multiplier and a accumulator. As mentioned before, we partition the on-chip cluster matrix, regression matrix, and training batch data, which makes the foundation for pipeline calculation of the matrix-matrix multiplication calculation. Due to pipeline, the M2MM complexity is reduced from $\mathcal{O}(K \cdot N \cdot T)$ to $\mathcal{O}(T + max\{K, N\})$. Here the pipeline stage is the chunk size T, the initiation interval (II) is the single PE execution time. In theory, the total execution time (T_{exec}) for one chunk of training batch ($N \times T$) inner product with cluster matrix ($K \times T$) will be: $T_{exec} = II \times (T + max\{K, N\})$.

D. Updating-Matrix Sparsity

Compared to the sequential regression model, one of the obstacles of HyDRAF is using the select matrix to update the cluster hypervector matrix BRAM. Let's revisit Equation (1). Although we can create a select matrix $\mathbf{S}$ efficiently in the

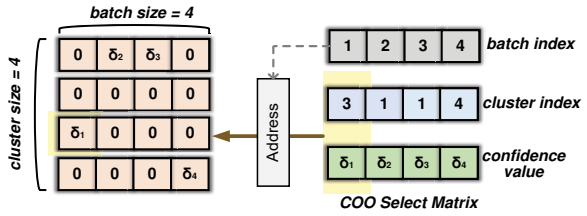

Fig. 5. COO representation of select matrix.

softmax layer, the way that the register or BRAM is used to store the select matrix is not efficient. Since for each data of training batch, it can only have one target cluster, which means for a $K \times N$ $\mathbf{S}$ matrix, there are only K 1s and the rest will be 0. The matrix $\mathbf{S}$ is a typical sparse matrix. Another problem of Equation (1) is that there are two M2MM operations involved, where one of them is sparse matrix-matrix multiplication(SpM2MM). Although FPGA has powerful parallel processing ability by deploying systolic array accelerator, we use coordinate list (COO) to store select matrix information and use only one matrix-matrix multiplication to finish cluster matrix update. As shown in Figure 5, we will have one cluster index array ($\mathbf{S}_i$) with size K. The index of the index matrix, $\mathbf{S}_i$, will be the address of training data in the batch, and the value of the array element will be the index of the cluster. For example, suppose 1^{th} member of the training batch has a corresponding cluster index of 3. In this way, $S_i[1] = 3$. We also need S_c to store the confidence value of the select matrix. It is obvious to see that the storing overhead or space complexity is reduced from $\mathcal{O}$(K×N) to $\mathcal{O}$(2×K). Besides reducing space complexity, using COO representation also reduces time complexity. We modify the Equation (1) below:

$$\mathbf{C}_{newi} = \mathbf{C}_{oldi} + \mathbf{H}_j \cdot \mathbf{S}_c[j] \cdot \mathbf{L}[j] \tag{2}$$

Here j is the index of data in training batch and its range is $[1, N]$. The i is the index of j's corresponding highest confidence, which is:

$$i = \mathbf{S}_i[j] \quad \text{for } j \in [1, N] \tag{3}$$

Based on (2) and (3), the new time complexity of updating cluster matrix BRAM for each chunk is reduced from $\mathcal{O}(T \times N + (T + max(N, K)))$ to $\mathcal{O}(T \times N)$. After using COO representation for sparse select matrix, both execution time and hardware resources utilization efficiency have been improved significantly.

V. Evaluation

A. Experimental Setup

We synthesize and implement our accelerator design on several different FPGA platforms with different available resources, including Xilinx Virtex UltraScale+ FPGA VCU118, Xilinx Alveo U200, and Xilinx Alveo U250. Specifically, we design and debug our accelerator on Xilinx Vitis HLS[16]. After generating the accelerator's RTL code from Vitis, we import it as an IP module into Vivado[17]. The system-level block design on Vivado is shown in Figure 6. HyDRAF access off-chip DRAM via AXI Interconnection IP. We use a

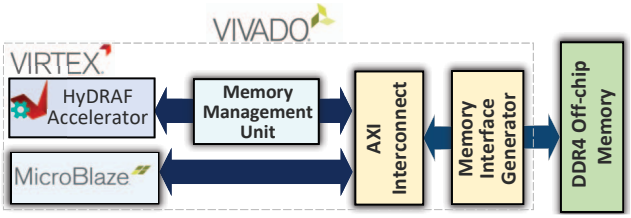

Fig. 6. HyDRAF system level design.

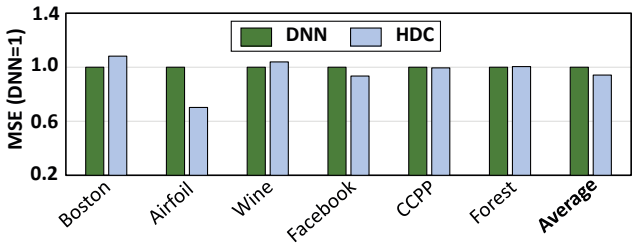

Fig. 7. Comparison of the mean square error (MSE) of regression between HDC-based regression model and DNN on different datasets when K = 8 and D = 2K bytes.

memory management unit (MMU) between HyDRAF and AXI IP to transfer memory address space from 64 bits to 32 bits. Since our accelerator is targeting to be deployed on an edge computing environment, the MicroBlaze CPU, a lightweight Xilinx softcore CPU widely used in embedded system design, is used to control the accelerator. We also compare the C++ version of our design on ARM Cortex A53 CPU and PyTorch for optimized implementation of HDC regression on NVIDIA 1080 GPU. One of the common problems that restricted FPGA accelerator development is the balance between resources utilization and performance.

B. Parameters and Datasets

There are five knobs in our design: single data precision (P), the number of chunks that we use to divide the hypervector, $\frac{D}{T}$, each chunk's size of T, the number of cluster K, and the size of batch N. Each parameter affects HDC regression accuracy and performance. We report the effectiveness of our approach in terms of both algorithm accuracy and hardware efficiency. The algorithm metric is validation loss, while the hardware metrics include throughput and energy consumption. We evaluate HyDRAF accuracy and efficiency on popular regression datasets, including Boston housing (Boston) [18], NASA airfoil self-noise (Airfoil) [19], wine quality prediction (Wine) [20], Facebook performance metrics (Facebook) [21], combined cycle power plant prediction (CCPP) [22], and forest fire prediction (Forest) [23].

C. Quality of Regression

Figure 7 compares HyDRAF's regression accuracy with the deep neural network (DNN). Here the DNN model is trained with Tensorflow[24]. Here we choose the mean square error (MSE) as the measurement of the model's regression error metrics. Less MSE represents higher regression quality. Our HDC-based regression model with configuration in Figure 7 shows relative regression accuracy compared with DNN.

TABLE I
QUALITY OF HDC REGRESSION USING DIFFERENT PRECISIONS, NUMBER OF CLUSTERS, AND THE DIMENSIONALITIES.

	8-bit Precision					16-bit Precision					Float Precision				
# clusters (K) \ Dimension (D)	***D=0.5k***	***D=1k***	***D=2k***	***D=3k***	***D=4k***	***D=0.5k***	***D=1k***	***D=1k***	***D=3k***	***D=4k***	***D=0.5k***	***D=1k***	***D=2k***	***D=3k***	***D=4k***
K=1	75.1	78.7	82.2	84.0	84.0	79.1	82.8	86.5	88.4	88.4	80.7	84.5	88.3	90.2	90.2
K=2	77.7	82.2	84.9	86.7	86.7	80.9	85.6	88.4	90.3	90.4	82.6	87.4	90.2	92.1	92.1
K=4	86.6	89.4	91.7	92.1	92.2	88.3	91.2	93.5	94.0	94.0	88.3	91.2	94.1	94.1	94.1
K=8	87.5	87.5	90.3	91.2	94.1	88.3	91.2	93.5	94.0	94.1	91.2	93.1	95.0	95.0	95.0
K=16	91.2	92.2	94.3	94.3	95.0	92.1	93.1	95.0	95.0	95.0	92.6	93.4	95.0	95.0	95.0

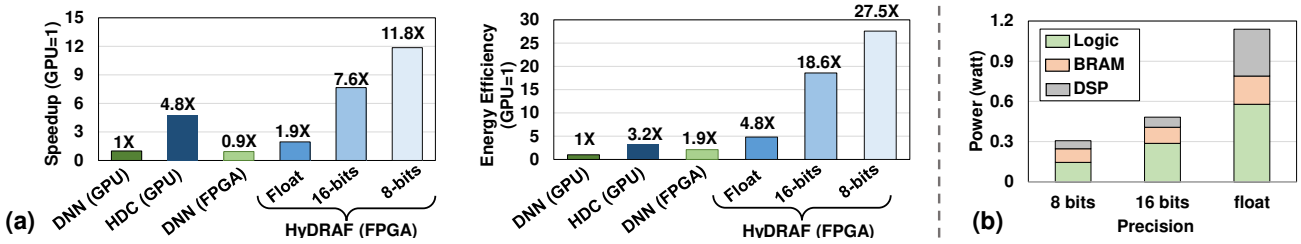

Fig. 8. Comparison of HyDRAF efficiency as compared to state-of-the-art DNN on GPU and FPGA.

The quality of HDC regression directly depends on the parameters. Table I shows the average quality of regression using different bit precision, dimensionality, and number of clusters. For precision, we exploit 8-bits, 16-bit, and floating-point. The number of clusters is also changing from $K = 1$ to $K = 16$. All results are reported using a batch size equal to $N = 8$. Our evaluation shows that HDC regression can provide maximum quality of using any precision. However, for low precision models, HDC requires a higher number of clusters to ensure maximum quality. For example, our regression ensures maximum accuracy using $K = 12$ clusters using the floating-point model. Using 8-bit precision, the same accuracy can be provided using $K = 16$ clusters.

In addition, Table I shows the impact of hypervector dimensionality on HDC regression accuracy. Similar to the number of clusters, HDC regression accuracy increases with the hypervector dimensionality. However, this accuracy saturates using a hypervector larger than $D = 3k$. For example, HDC regression using $D = 2k$ and $D = 512$ only provide 0.7% and 3.8% lower quality as compared to regression with full dimensionality of $D = 4k$ ($K = 16$ and 8-bit precision model).

Dimensionality is also in trade-off with the number of clusters. To provide the same quality or regression, one can select to use a high-dimensional model with a lower number of clusters ($D \gg$ and $K \ll$) or a higher number of clusters with lower dimensionality.

D. Efficiency vs. State-of-the-art

Figure 8 shows the performance speedup and energy efficiency of HyDRAF running on GPU and FPGA. The results also compare HyDRAF efficiency with state-of-the-art HDC implementation [4] and state-of-the-art DNN accelerators running on GPU and FPGA. We used DNNWeaver V2.0 [25] for efficient implementation of the NN inference, and FPDeep [26] for NN training on a single FPGA device. FPGA implementations are optimized to maximize performance by utilizing FPGA resources. All results listed in Figure 8 are relative to DNN performance and energy efficiency. Note that we compare the baseline DNN and HDC regression since several existing optimizations, e.g., model binarization or pruning [27], can be applied to both methods. During training, HyDRAF achieves, on average, 12.6× faster and 14.1× more energy-efficient computation than FPGA-based DNN implementation, respectively. The high efficiency of HyDRAF in training comes from: (i) HyDRAF capability in creating an initial model that significantly lowers the number of required retraining iterations. (ii) It eliminates the costly gradient descent for the model update. This results in a higher HyDRAF efficiency, even in terms of a single training iteration.

Figure 8 also compares HyDRAF efficiency using different bit precision. In each bit precision, the hypervector dimensionality is set to ensure HyDRAF provides maximum accuracy. For instance, HDC regression exploits hypervectors with $D = 512$, $D = 1k$, and $D = 2k$ for models with floating point, 16-bits, and 8-bits precision, respectively. Our evaluation shows that our FPGA acceleration provides maximum throughput and efficiency when using low-precision (and high-dimensional) vectors. At the same time, GPU is more effective in dealing with floating-point (and low-dimensional) vectors. Particularly, using 8-bit precision, HyDRAF supports regression operation using efficient LUTs, while FPGA needs to rely on limited and costly Digital Signal Processor (DSP) blocks to support floating-point operations. Our results indicate that HyDRAF using 8-bits precision can provide 6.1× and 5.7× (1.5× and 1.4×) speedup and energy efficiency compared to floating-point (16-bits) models, respectively.

We also provide an FPGA kernel power breakdown in Figure 8c to make a comparison of the power efficiency for different data precision. The power estimator tool that we used is the Vivado power estimator, and the targeting device is Xilinx Virtex Ultrascale+ XCU200 FPGA. It is pretty obvious to see that, compared to fixed-point operation, the floating-point operation uses more LUTs and more DSPs when accessing on-chip storage and carrying multiplication-addition operation. Therefore, floating-point based operations consume more power compared to fixed-point operations. Therefore,

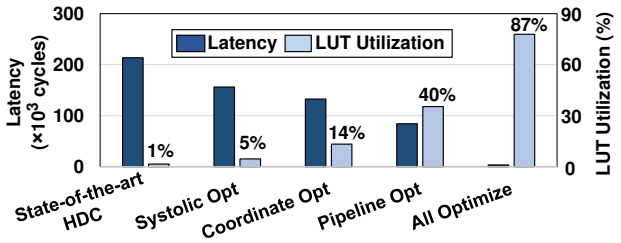

Fig. 9. Impact of different optimization techniques on HyDRAF resource utilization and latency (single batch).

it makes sense that the energy efficiency will decrease when doing a higher precise mathematical process.

E. HyDRAF Optimizations and Comparison

Figure 9 shows the impact of different HyDRAF optimizations on the performance speedup and energy efficiency improvement of HyDRAF. All results are compared to DNN running on GPU. For HDC, we use the state-of-the-art regression implementation in [4] as the baseline. HyDRAF results are reported for three main optimizations: (1) using systolic array, (2) coordinate list, (3) pipeline the systolic array.

To ensure high computation efficiency, our platform relies on FPGA LUTs. However, the baseline FPGA implementation can often occupy around 1% of the FPGA LUT resource. Each of our optimizations aims to increase LUT utilization and provide higher computational throughput. For example, coordinate index optimization enhances the LUT utilization from 1% to 14% by parallel the cluster model updating process and avoid redundancy M2MM calculation. Similarity, pipelining the systolic array improves LUT resource utilization to 40% by using more LUTs to parallel on-chip storage access and matrix-matrix multiplication, respectively. Our evaluation shows that HyDRAF with all optimizations achieves 87% resource utilization, improving the latency by 63× compared to the baseline implementation.

F. Resource Utilization & Performance Trade-off

The performance of our FPGA implementation has a direct relation with on-chip resources. The following four types of resources are important for FPGA acceleration: block random access memory (BRAM), LUTs, Flip Flops (FF), and DSP. Here, we first analyze the main performance bottleneck of our acceleration and then illustrate how to choose the suitable configurations based on the performance requirement.

Tune HyDRAF Parameters: For our regression model with batch parallelism, the most resources utilizing processes are: (1) Similarity check and (2) Inner product, as both require large-scale matrix-matrix multiplication. To maximize the parallelism, we partition the on-chip storage based on the matrix's size. When the hypervector size (D) is constant, the four important knobs to determine the size of the accelerator are: batch size (N), regression model and cluster size (K), the chunk size (T), and the data precision (P). Our framework enables users to tune the parameters based on their desired metrics. Increasing the value of N will bring faster training. Larger K and P values improve the quality of regression. However, we limited these two parameters to $k = 8$ and $P = 8$ to ensure maximize computation efficiency.

TABLE II
HyDRAF Resources Utilization and performance with default parameter of Off-chip Encoding Version

Dimension (D)	0.5K	1k	2k	3k	4k
LUTs	578898	643046	778774	1100240	1678453
FF	247713	349441	542635	732119	1174445
DSP	76	76	76	78	78
FPGA Boards	Alveo U200	Alveo U200	Alveo U200	Alveo U250	VCU118
$\mathbf{L_{train}}$(cycle)	1503	2136	3422	5501	6034
$\mathbf{L_{infer}}$(cycle)	53	81	113	144	177

Chunk Size: Figure 10a shows the latency and resource utilization of FPGA when the chunk sizes varies from $T = 16$ to $T = 128$. As we expect, using a larger chunk size increases resource utilization and throughput. However, the latency improvement does not linearly scale with resource utilization. A large chunk size significant resources the overhead of pipelining. We observe that using 64 chunk size provides maximum throughput improvement efficiency. However, further increasing the chunk size significantly increases the resource utilization while having a minor impact on HyDRAF latency. Our evaluation shows that using 128 chunk size has 1.9× lower latency improvement per resource utilization than with 64 chunks. Note that HyDRAF with chunk size larger than $T = 64$ cannot fit inside our Alveo 200 accelerator card.

Resource Utilization: Table II reports the details of our FPGA implementation: latency and resources utilization when hypervector size varies from $D = 512$ to $D = 4k$. In all experiments, the BRAM utilization is less than 5%. Our results show that using a larger hypervector size increases the FPGA resource utilization. Table II lists the FPGA board that can fit our regression model without compromising the performance. For all configurations, the clock cycle is 7.3 ns. Our evaluation on an Alveo U200 Accelerator Card platform drawing less than 4 Watt for kernel Xilinx Virtex Ultrascale+ XCU200 FPGA and 41.7 Watt for the whole board shows that HyDRAF can process 1.2 million classifications per second during inference. During training, the throughput is 0.32 million data per second at each batch.

Our evaluation also shows that, unlike our expectation, the FPGA latency increases linearly with hypervector dimensionality. We observe that the increase in the latency comes from loading a larger amount of encoded data from I/O. One solution to resolve this is to load original data and perform on-chip encoding, rather than storing and loading large encoded hypervectors from off-chip DRAM. We will discuss more details of encoding online in section V-G.

G. HyDRAF Encoding On-chip vs. Off-chip

Due to AXI I/O bandwidth limitation and on-chip partitioning, storing and loading access time increase significantly with hypervector dimensionality. We address this issue by enabling on-chip encoding, storing and loading original data from off-chip DRAM. Figure 10b compares HyDRAF latency during off-chip and on-chip encoding when the hypervector dimensions varies from $D = 512$ to $D = 4k$. Our evaluation shows that using a low-dimensional hypervector, it is more desirable to perform off-chip encoding as I/O cost is minimal. However, as hypervector size is growing, the on-chip encoding outperforms the off-chip method. Although on-chip encoding

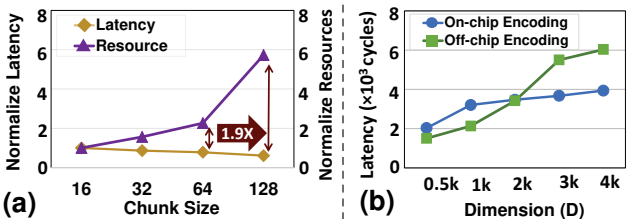

Fig. 10. On & Off-chip Encoding's affect on training latency.

pays an extra cost for repeated data mapping, it eliminates the cost of loading large hypervectors, thus eliminating the I/O from being the computational bottleneck.

VI. Related Work

Hyperdimensional computing (HDC) has been proved to be successful in multiple cognition tasks [28], [29]. [4] is the first work proposed to handle regression tasks based on multi-model HDC. Due to HDC's hardware-friendly operation, multiple hardware accelerators have been investigated, including designing new ASIC [30], exiting FPGA [12], [14], [31], and processing in memory (PIM) architectures [32], [33]. However, all these accelerators are targeting classification tasks. Although [4] proposed a new HDC based regression framework, it is not efficient on FPGA since it didn't fully utilize FPGA hardware resources and didn't consider regression hypervector matrix's sparsity problem. Our approach: HyDRAF, based on the previous HDC regression algorithm, is optimized considering FPGA on-chip resources, memory boundary, and matrix sparsity, and is proven to be successful when handling regression tasks. Besides, compared to [4] which relied on CPU to implement HD hypervector encoding process, we successfully integrate it into our on-chip accelerator design to break the memory boundary and realize online learning.

VII. Conclusion

In this paper, we propose an FPGA acceleration of hyperdimensional regression supporting online learning. To overcome the computation overhead resulting from the long size of the hypervector, we introduce multiple FPGA optimizations to efficiently handle long vector access, such as on-chip storage partitioning. Furthermore, we optimize the model update process by using efficient sparse matrix representation. We also integrate the encoding module into the accelerator to realize online training by reducing off-chip DRAM access.

VIII. Acknowledgements

This work was supported in part by National Science Foundation #2127780, Semiconductor Research Corporation (SRC) Task #2988.001, Department of the Navy, Office of Naval Research, grants #N00014-21-1-2225 and #N00014-22-1-2067, the Air Force Office of Scientific Research under award #FA9550-22-1-0253, and a generous gift from Cisco.

References

[1] X. Qiu *et al.*, "Ensemble deep learning for regression and time series forecasting," in *CIEL*, pp. 1–6, IEEE, 2014.
[2] G. Huang *et al.*, "Machine learning for electronic design automation: A survey," *TODAES*, vol. 26, no. 5, pp. 1–46, 2021.
[3] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *CC*, 2009.
[4] A. Hernández-Cano *et al.*, "Reghd: Robust and efficient regression in hyper-dimensional learning system," in *ACM/IEEE DAC*, pp. 7–12, IEEE, 2021.
[5] A. Rahimi *et al.*, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *ISLPED*, pp. 64–69, ACM, 2016.
[6] G. Karunaratne *et al.*, "Robust high-dimensional memory-augmented neural networks," *Nature communications*, vol. 12, no. 1, pp. 1–12, 2021.
[7] Hernandez-Cane *et al.*, "Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system," in *DATE 2021*, pp. 56–61, IEEE, 2021.
[8] H. Li *et al.*, "Hyperdimensional computing with 3d vrram in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *IEDM*, pp. 16–1, IEEE, 2016.
[9] Z. Zou *et al.*, "Scalable edge-based hyperdimensional learning system with brain-like neural adaptation," in *ACM SC*, pp. 1–15, 2021.
[10] S. Datta *et al.*, "A programmable hyper-dimensional processor architecture for human-centric iot," *JETCAS*, vol. 9, no. 3, pp. 439–452, 2019.
[11] B. Khaleghi *et al.*, "Shear er: highly-efficient hyperdimensional computing by software-hardware enabled multifold approximation," in *ISLPED*, pp. 241–246, 2020.
[12] M. Imani *et al.*, "Revisiting hyperdimensional learning for fpga and low-power architectures," in *HPCA*, IEEE, 2021.
[13] M. Schmuck *et al.*, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *arXiv preprint arXiv:1807.08583*, 2018.
[14] Morris *et al.*, "Adaptbit-hd: Adaptive model bitwidth for hyperdimensional computing," in *ICCD*, pp. 93–100, IEEE, 2021.
[15] M. Imani *et al.*, "Neural computation for robust and holographic face detection," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 31–36, 2022.
[16] V. Kathail, "Xilinx vitis unified software platform," in *ACM/SIGDA FPGA*, 2020.
[17] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.
[18] "Boston Housing Dataset." http://lib.stat.cmu.edu/datasets/boston.
[19] R. L. Gonzalez, *Neural networks for variational problems in engineering*. PhD thesis, Universitat Politècnica de Catalunya (UPC), 2009.
[20] P. Cortez *et al.*, "Modeling wine preferences by data mining from physicochemical properties," *Decision Support Systems*, vol. 47, no. 4, pp. 547–553, 2009.
[21] S. Moro *et al.*, "Predicting social media performance metrics and evaluation of the impact on brand building: A data mining approach," *JBR*, 2016.
[22] P. Tüfekci, "Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods," *JEPE*, 2014.
[23] P. a. Cortez, "A data mining approach to predict forest fires using meteorological data," 2007.
[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[25] H. Sharma *et al.*, "From high-level deep neural models to fpgas," in *MICRO*, pp. 1–12, IEEE, 2016.
[26] T. Geng *et al.*, "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," in *FCCM*, pp. 81–84, IEEE, 2018.
[27] G. Yuan *et al.*, "An ultra-efficient memristor-based dnn framework with structured weight pruning and quantization using admm," in *ISLPED*, pp. 1–6, IEEE, 2019.
[28] P. Poduval, Z. Zou, X. Yin, E. Sadredini, and M. Imani, "Cognitive correlative encoding for genome sequence matching in hyperdimensional system," in *2021 58th ACM/IEEE DAC*, pp. 781–786, IEEE, 2021.
[29] Z. Zou, Y. Kim, M. H. Najafi, and M. Imani, "Manihd: Efficient hyperdimensional learning using manifold trainable encoder," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 850–855, IEEE, 2021.
[30] M. Eggimann *et al.*, "A 5 μw standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing," *IEEE TCAS I*, 2021.
[31] Y. Hao *et al.*, "Stochastic-hd: Leveraging stochastic computing on hyperdimensional computing," in *ICCD*, pp. 321–325, IEEE, 2021.
[32] A. Kazemi *et al.*, "Mimhd: Accurate and efficient hyperdimensional inference using multi-bit in-memory computing," in *ISLPED*, pp. 1–6, IEEE, 2021.
[33] Z. Zou *et al.*, "Biohd: an efficient genome sequence search platform using hyperdimensional memorization," in *ISCA*, pp. 656–669, 2022.