Scalable and Interpretable Brain-Inspired Hyper-dimensional Computing Intelligence with Hardware-software Co-design

Hanning Chen, Yang Ni, Wenjun Huang, and Mohsen Imani
Department of Computer Science, University of California, Irvine
{hanningc, yni3, wenjunh3, m.imani}@uci.edu

Abstract—During the advancement of modern deep learning algorithms, models become increasingly demanding in computing resources and power-hungry, such that they are considered less hardware-friendly for many real-world deployments. The motivation behind brain-inspired computing, or neuromorphic computing, is that the human brain remains the most sophisticated yet efficient learning module ever. We focus on HyperDimensional Computing (HDC), which aims to realize efficient learning via brain-like high-dimensional vector operations. Prior research works have shown that HDC is a lightweight alternative to deep learning in various applications, such as classification and reinforcement learning. HDC can also serve as a reasoning machine on graph datasets and an efficient information retrieval method for genomic sequencing. In this paper, we revisit hardware-software codesigns of HDC, covering the latest developments in both HDC algorithms and accelerator designs. We also carried out extensive comparisons between HDC works and the state-of-the-art.

1. Introductions

The Machine Learning (ML) algorithm has become a powerful tool for computers to perform cognitive tasks, as it has shown super-human accuracy in areas such as image classification, speech recognition, and robotic control. However, many existing ML algorithms, especially those based on the Deep Neural Network (DNN), are computationally intensive, have limited flexibility in parallelization, and require high-precision arithmetic operations. In the era of the Internet of Things (IoT), these limitations pose significant challenges for implementing ML on resource-constrained devices, including embedded systems and edge devices. Therefore, we have seen an increasing amount of research in the emerging field known as neuromorphic computing, focusing on more efficient ML approaches that are inspired by brain structure and functionalities.

As one of the brain-inspired methods, HyperDimensional Computing (HDC) seeks to overcome these challenges by mimicking the high-dimensional nature of data processing in the brain [2], [33]. The development of the HDC paradigm is based on interdisciplinary research at the intersection of theoretical neuroscience and computer science. For example, human short-term memory carried out in the hippocampus region can be modeled as an auto-associative memory based on orthogonal representations, which motivates the high-dimensional representation and light-weight memorization in HDC. More specifically, an HDC encoder maps original inputs to a high-dimensional space (also known as the hyperspace) by representing them as long binary (or real-valued) vectors of several thousand dimensions, i.e., hypervectors. Then, to realize memorization as well as other brain-like functionalities, HDC uses simple element-wise operations to perform symbolic manipulations on these hypervectors, such as bundling (More details in Section 2).

HDC owns several unique properties that have been crucial for ML applications with stringent efficiency requirements [1], [8]. For example, the hypervector representation is holographic, meaning that information is distributed evenly across vector components. Holographicness plays a key role in encoding distinct concepts with random and near-orthogonal representations, allowing HDC to represent and manipulate atomic symbols in an intuitive way. Another property is that HDC supports concise and efficient learning and reasoning, thanks to both the operation-wise simplicity and the richness of the hypervector representation. These characteristics lead to several advantages of HDC over conventional ML methods, such as lower power/energy consumption, faster learning ability, and better model interpretability [10], [12], [16], [22].

HDC-based algorithms have been applied to various domains of applications as a more efficient alternative to existing solutions. For classification tasks, prior HDC works have proposed solutions to the recognition of text, speech, and image, reaching significantly faster training and inference with comparable learning quality to methods like DNN and SVM. Besides those common data modalities, HDC has also been leveraged for processing multi-sensory readings and bio-signals like Electroencephalography (EEG) and Electro-myography (EMG) [15],[19]. More importantly, HDC enables single-pass classifier training with satisfying quality, which means that it can learn from data in one pass without the need for multiple iterations. Recent works also propose HDC-based algorithms for regression and Reinforcement Learning (RL), where hypervectors are used to learn function representations with fewer samples and iterations [14], [18], [24], [25]. Compared to DNN-based algorithms, they bring notably faster convergence, resulting in a more efficient RL agent for robotic control and resource management. Apart from learning tasks, HDC has also shown its strength in reasoning tasks such as graph reconstruction, node classification, and graph matching, where the hypervector representation powers a transparent, interpretable, and lightweight reasoning process [4], [29]. Finally, HDC is bestowed with brain-like efficiency in pattern-oriented computations, making it a great fit for information retrieval in tasks like genome sequencing [5], [9], [26]. In HDC-based sequencing methods, a single hypervector can effectively combine multiple patterns, thereby giving much higher efficiency in similarity computation and matching.

Adding onto the efficiency advantage shown at the algorithm level, HDC is especially more hardware-friendly during deployments, compared to the DNNs with deep layers and backprop-based training. HDC can be implemented in a resource-effective manner since the hypervectors are often represented with low-precision components, e.g., binaries, and highly parallelizable as most operations in the hyperspace are dimension-independent. These nice properties allow HDC to further enhance its performance by exploiting the characteristics of different hardware platforms, enabling HDC-based algorithms to run on low-power and resource-limited devices that are suitable for real-time and online applications [20], [21].

For example, HDC can greatly benefit from the high parallelism offered by Processing-in-Memory (PIM) architectures [27], [31], [32], where it tackles the von Neumann bottleneck on both the hardware and algorithm levels. Although PIM platforms have tight budgets for the maximum supported model precision, we can leverage the robustness of HDC-based algorithms on low-precision models even after aggressive quantization. On the other hand, the reconfigurable computing logic in FPGA enables the specific data path design targeting HDC models. As shown in previous works [?], HDC is also intrinsically suitable for acceleration on FPGAs because of its natural algorithm-level parallelism and loose requirement for high-quality yet power-hungry computing logic (e.g., digital signal processing (DSP) unit).

In this paper, we summarize prior hardware-algorithm codesigns of HDC accelerator targeting different machine leaning applications. These applications cover a wide range, including classification, regression, reinforcement learning, graph reasoning, and genomic sequence matching. For each application, we introduce both HDC model design and accelerator design. We also provide hardware performance results which include resource utilization and comparison with state-of-the-art works.

2. HDC Motivation and Basics

HDC draws inspiration from the insights gained from neuroscience, specifically, the observation that the human brain consists of approximately 100 billion neurons and an astonishing 1000 trillion synapses. Consequently, it becomes plausible to represent all possible states of a human brain through a high-dimensional vector. To closely mimic the information representation and memorization mechanisms observed in the human brain, HDC translates low-dimensional inputs into a hyperspace. The hyperspace comprises hypervectors, each possessing over a thousand independent and identically distributed (*i.i.d.*) components, which can take the form of binary, integer, real, or complex values [34]. In this section, we

2024 IEEE Custom Integrated Circuits Conference (CICC) | 979-8-3503-9406-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/CICC60959.2024.10529049

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.
979-8-3503-9406-1/24/$31.00 ©2024 IEEE

provide the fundamental background for understanding HDC systems.

A general HDC system can roughly be divided into three stages. The first stage in HDC is mainly concerned with the **Encoding** process, which entails the mapping of data into a high-dimensional space and the generation of atomic hypervectors. Subsequently, HDC performs a series of **Hypervector Operations** on these hypervectors, which are categorized into *Binding* (Multiplication), *Bundling* (Accumulation), and *Permutation* (Shift-Rotate). For instance, when training HDC class hypervectors, all related hypervectors belonging to the same class are bundled together to form a unique class hypervector, which serves as the prototype for that data class. Finally, the third stage encompasses **Similarity Measurement**, where the similarity between the testing query hypervector and the class hypervectors is assessed. Depending on the specific task at hand, a range of hyperspaces has been proposed, spanning from binary to complex-valued hypervectors. Here we take binary hypervector encoding as an example. More variations are discussed in the subsequent sections. Binary or Bipolar HDC encoding systems primarily rely on vectors with a length of up to 10^4 bits, where these vectors comprise values of +1 (representing logic 1) and -1 (representing logic 0). Recent efforts have been directed toward reducing the dimensionality of these hypervectors to enhance application accuracy, recognizing that longer hypervectors carry a more information-rich payload. In addition to dimensionality, the type of encoding employed is another crucial factor that directly affects accuracy.

A. Hypervector Encoding

1) Binary/Bipolar HDC Encoding

In binary HDC, data is typically encoded into random hypervectors, with an emphasis on ensuring *orthogonality* between them. The concept of orthogonality plays a pivotal role in HDC, as it allows for the effective representation of unique features or symbols. For example, it enables the representation of letters in a text processing system, pixel positions in an image in a cognitive task [35], or time series in a voice recognition task [37]. Notably, randomly generated vectors tend to exhibit a degree of near orthogonality to each other [10]. Consequently, these randomly generated and pre-allocated vectors effectively serve as atomic data primitives within HDC systems, representing symbols. Various methods have been employed in the literature to achieve near orthogonality among hypervectors. One prominent technique involves initiating the process with an initial seed vector and subsequently determining additional vectors through random bit-flip operations [38].

For example, this encoding is applied for data in the form of feature vectors (address and value). For input data of d dimensions, the model generates an address codebook with one entry for each dimension $\{A_1, \cdots, A_d\}$. And a value codebook $\{L_1, \cdots, L_q\}$ corresponding to the q quantized levels of the continuous value (we assume that the data is normalized for each dimension and thus shares the same range). The encoder then performs a bundling of the address-value pair association for a data point $\boldsymbol{x} \in \mathbb{R}^d$:

$$H_{\boldsymbol{x}} = \sum_{i=1}^{d} A_i \odot V_i, V_i \in \{L_1, \cdots, L_q\} \quad (1)$$

2) Kernel-based HDC Encoding

Prior works [28] also exploit an encoder method inspired by the Radial Basis Function (RBF) kernel trick [42, 43], for mapping data points into the hyperspace. This encoder considers the non-linear relation between the features during the encoding. However, kernels like RBF implicitly maps inputs to an infinite-dimensional space, and the exact mapping is intractable. Prior work in [39] proposes that with a large but finite dimensional mapping Z, the shift-invariant kernel K can be approximated using inner-products:

$$K(x_m - x_n) \approx Z_D(x_m)^T Z_D(x_n) \quad (2)$$

where D is the dimensionality of the mapping. To approximate the RBF kernel, the mapping is defined as follows:

$$Z_D(x) = \sqrt{\frac{2}{D}} cos\left(\vec{H}x + \vec{B}\right) \quad (3)$$

$\vec{H}$ is a vector of dimension D with its elements randomly sampled from standard Gaussian distribution $\mathcal{N}(0,1)$ and $\vec{B}$ functions as a bias vector with elements sampled from uniform distribution $\mathcal{U}(0,2\pi)$. Once they are randomly generated, we keep them fixed during the later learning and inference.

B. Hypervector Operations

Once information is represented within the hyperspace, composite representations can be constructed through dimension-independent operations. These operations include bundling, binding, and permutation, and they maintain the dimensionality of the hypervectors, resulting in a hypervector residing in the same hyperspace as the original operands. Furthermore, these operations can be combined, offering versatility in creating customized encodings that cater to various applications, effectively capturing the inherent compositionality found within the data.

1) Bundling

Bundling, also known as point-wise addition or accumulation, calculates a hypervector $\mathcal{Z} = \Sigma_i \mathcal{H}_i$ from a set of input hypervectors $\{\mathcal{H}_1, \mathcal{H}_2, \cdots, \mathcal{H}_i\}$. In comparison to randomly generated hypervectors, the resulting $\mathcal{Z}$ is maximally similar to the inputs $\{\mathcal{H}_1, \mathcal{H}_2, \cdots, \mathcal{H}_i\}$. In the realm of high-dimensional space, bundling functions as a memory operation and offers a straightforward means to verify the presence of a query hypervector within a bundled set.

2) Binding

Binding, also referred to point-wise multiplication, serves the purpose of establishing connections between two related hypervectors. Hypervectors $\mathcal{H}_1$ and $\mathcal{H}_2$ are bound together to form $\mathcal{Z} = \mathcal{H}_1 * \mathcal{H}_2$ which is approximately orthogonal to both $\mathcal{H}_1$ and $\mathcal{H}_2$. Due to reversibility, in bipolar cases, $\mathcal{Z} * \mathcal{H}_1 = \mathcal{H}_2$, allowing for the retrieval of information from both hypervectors through the bound hypervector.

3) Permutation

The unary operation unique to HDC, known as Permutation (ρ^i), involves the rotational shifting of a hypervector, with i indicating the number of times the operation is applied. When applied to a hypervector $\mathcal{H}_1$, this operation returns a dissimilar hypervector $\mathcal{Z} = \rho(\mathcal{H}_1)$ to the input. It also allows for the assignment of specific orders to hypervectors within the hyperspace. Notably, the inverse operation $\rho^{-i}(\mathcal{H})$ enables the exact retrieval of the original input hypervector, ensuring reversibility in the permutation process, as demonstrated in \cite{cohen2018bringing}. The permutation operator in HDC provides a flexible means of manipulating and organizing hypervectors, which proves highly beneficial in a variety of cognitive computing tasks.

C. Similarity Measurement

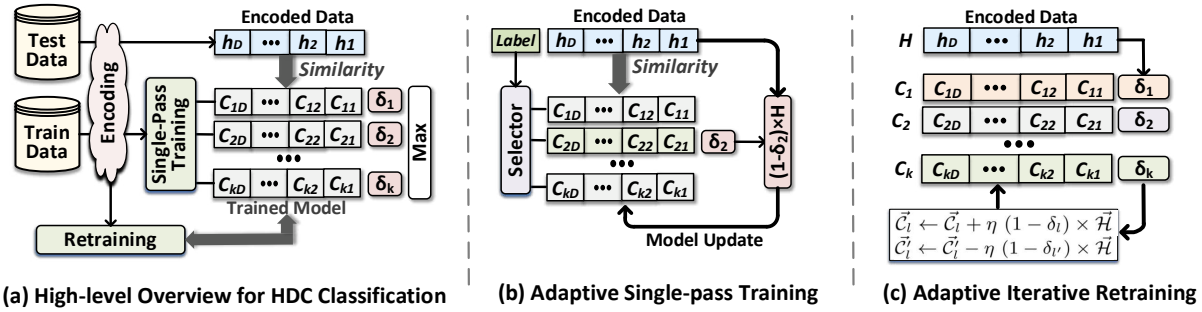
In HDC, the assessment of similarity between hypervectors is a pivotal measure, determined by the function $\delta(\mathcal{H}_1, \mathcal{H}_2) \rightarrow \mathbb{R}$. This similarity metric is instrumental in discerning the relationships between hypervectors. It quantifies the angle between two hypervectors and, depending on the type of data, various implementations of this measure can be applied. For real or integer data, cosine similarity is commonly used. Binary representations, on the other hand, can efficiently utilize the Hamming distance for this purpose.

In the case of non-binary hypervectors, cosine similarity, defined by Eq. (1) is used to measure their similarity. This measure focuses on the angle between the hypervectors and disregards the influence of their magnitude, where $|\cdot|$ signifies the magnitude. Unlike the inner product operation of two vectors, which affects both magnitude and orientation, cosine similarity solely depends on the orientation. In most high-dimensional algorithms featuring non-binary hypervectors, cosine similarity is more commonly utilized than the inner product. Moreover, when $cos(\mathcal{H}_1, \mathcal{H}_2)$ approaches 1, it signifies an exceedingly high level of similarity. For example, $cos(\mathcal{H}_1, \mathcal{H}_2) = 1$ indicates two hypervectors $\mathcal{H}_1$ and $\mathcal{H}_2$ are identical. Conversely, when $cos(\mathcal{H}_1, \mathcal{H}_2) = 0$, the two vectors are considered dissimilar.

$$cos(\mathcal{H}_1, \mathcal{H}_2) = \frac{\mathcal{H}_1 \cdot \mathcal{H}_2}{|\mathcal{H}_1||\mathcal{H}_2|} \quad (4)$$

In the case of binary hypervectors with a dimensionality of D, where their components are either 0 or 1, the normalized Hamming distance, as computed in Eq. (2) is used to measure their similarity. When the Hamming distance between two hypervectors approaches

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.



(a) High-level Overview for HDC Classification (b) Adaptive Single-pass Training (c) Adaptive Iterative Retraining

Fig. 1. Hyperdimensional classification algorithm with single-pass and iterative training.

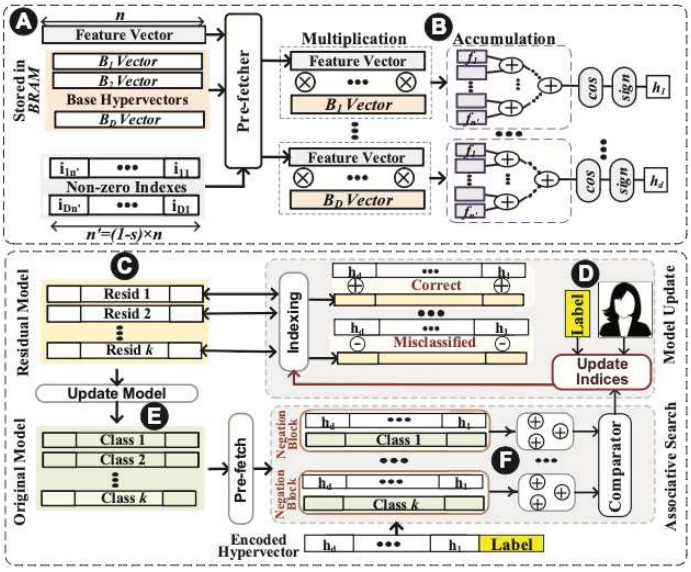


Fig. 2. Hyperdimensional Classification on FPGA.

0, they are defined as similar. For example, $Hamming(\mathcal{H}_1, \mathcal{H}_2) = 0$ indicates every single bit at each position is the same, marking $\mathcal{H}_1$ and $\mathcal{H}_2$ are identical. And when $Hamming(\mathcal{H}_1, \mathcal{H}_2) = 1$, it signifies that $\mathcal{H}_1$ and $\mathcal{H}_2$ are diametrically opposed, indicating maximum dissimilarity.

$$Hamming(\mathcal{H}_1, \mathcal{H}_2) = \frac{1}{D}\Sigma_i^D 1_{\mathcal{H}_1(i) \neq \mathcal{H}_2(i)} \quad (5)$$

3. HDC-BASED CLASSIFICATION

A. Hyperdimensional classification algorithm

In a classification problem, HDC first uses an encoder to map all training samples to the hyperspace in training, and similarly we encode the test samples at the beginning of the inference process. Then hypervectors associated with each class are bundled together during the training process, thereby enabling single-pass training. Compared with traditional DNN-based models, HDC models use much less power and have lower latency. Since HDC efficiently leverages hypervectors to compress and memorize the samples seen to class hypervectors and then perform learning tasks, HDC models are generally smaller than DNN model. In this section, we will go through a series of procedures in HDC-based classification algorithm. Fig. 1 (a) shows an overview of hyperdimensional classification.

1) Adaptive Single-pass Training:

In HDC classification, the naive hypervector addition results in saturation of class hypervectors by data points with the most common patterns. Due to model saturation, data points with non-common patterns are likely to miss-classified by the model.

As shown in Fig. 1 (b), with an adaptive training framework in OnlineHD [23], however, we can achieve efficient and accurate HDC learning in one pass. Instead of naively combining all encoded data, our approach adds each encoded data to class hypervectors depending on how much new information the pattern adds to class hypervectors. If a data point already exists in a class hypervector, we will add no or a tiny portion of data to the model to eliminate hypervector saturation. If the prediction matches the expected output, no update will be made to avoid overfitting. Assume that δ is the similarity between the encoded query hypervector and the hypervector for class C. Instead of naively adding data point to the model, we update the model based on the δ similarity. For instance, if a training sample that belongs to class 2 is wrongly classified to another class. Then, we will update the second class hypervector using the query hypervector weighted by the similarity difference: $(1 - \delta_2)\vec{H}$.

2) Adaptive Hypervector Retraining:

Although single-pass training is suitable for fast and ultra-efficient learning, embedded devices may have enough resources to perform more accurate learning tasks. HDC classifier also supports retraining to enhance the quality of the model. Instead of starting to retrain from a naive initial model, the retraining starts from the initial adaptive model. This enables HDC to retrain the model with a much lower number of iterations, resulting in fast convergence. Fig. 1 (c) shows the functionality during adaptive retraining.

The adaptive retraining follows a similar learning procedure as initial single-pass training. For each encoded training data point, say $\vec{H}$, we check the similarity of data with all class hypervectors in the model and updates the model for each miss-prediction. Retraining examines if the model correctly returns the label l for an encoded query $\vec{H}$. If the model mis-predicts it as label l', the model updates as the equation shown in the figure, where $\delta_l = \delta(\vec{H}, \vec{C_\ell})$ and $\delta_{l'} = \delta(\vec{H}, \vec{C_{\ell'}})$ are the similarity of data with correct and miss-predicted classes, respectively. This ensures that we update the model based on how far a train data point is miss-classified with the current model. In the event of a very far miss-prediction, the retraining makes a major change in the mode. While in case of marginal miss-prediction, the update makes smaller changes to the model.

B. Hardware and Software Co-Design

Fig. 2 presents the FPGA acceleration architecture of HDC classifier. We first encode each data point by computing the inner product of a feature vector with different weight vectors. Since the Gaussian distribution creates many near-zero values, we can easily create sparse random vectors to reduce the number of multiplications. The weight vectors can be stored with a vector with $(1-s) \times n$ consecutive non-zero values, and an index value that represents the index of the first non-zero element where s is the sparsity factor and n is the number of features. All weight vectors are stored in Block RAM (BRAM), which is on-chip FPGA memory (Fig 2. (a)). During the encoding, our approach prefetches the weight vectors from the BRAM blocks and stores them in the locally distributed memory that can be accessed faster than BRAM. During the encoding, FPGA reads the first m features of original data points (m $\leqslant$ n). Next, it accesses the weight vector and then multiplies $(1-s) \times n$ continuous dimensions of the feature vector with the corresponding weight vector. These multiplications are processed using Digital Signal Processor (DSP) blocks, and they are parallelized for different weight vectors. The results of the inner products are accumulated using a tree-based adder structure (Fig. 2. (b)). Finally, the cosine function is calculated using the lookup table (LUT) logic. Finally, the encoded hypervector can be binarized by considering the sign of the encoded data as a binary output.

C. Results

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.

We evaluate HDC classification accelerator on Xilinx KC705 FPGA. Table I presents the FPGA resource utilization. Regarding the acceleration efficiency, we compare the FPGA accelerator with Raspberry Pi 3B+ using ARM CPU in Table II. Table II lists the number of required OnlineHD dimensions in each bit precision that results in maximum classification accuracy. Table II also reports the average Energy-Delay Product (EDP) of FPGA and CPU running OnlineHD using different hypervector precision. All results are normalized to CPU EDP using hypervectors with 32-bit precision. Our evaluation shows that the CPU provides the highest efficiency using lower-dimensional vectors. This is because CPUs take the same number of resources to perform 1-bit or 8-bit arithmetic operations. This limits the amount of parallelism in the CPU. In contrast, FPGAs are more efficient in processing high-dimensional but low precision vectors. The lookup table and flip-flops resources on FPGA can perform several parallel bitwise operations and enable fast and efficient OnlineHD computation. Our goal is to maximum FPGA throughput, where we can process the maximum number of data points at a time. To eliminate off-chip memory to be a bottleneck of computation, FPGA needs to perform maximum computation over each read. We observe that FPGA provides minimum EDP using 2-bit precision. In this precision, OnlineHD maximizes FPGA resources while avoiding high precision arithmetic, as the complexity FPGA arithmetic increases exponentially with the bit-width.

Table I. Hyperdimensional Classification Acceleration on Xilinx KC705

	LUT	FF	BRAM	DSP
Utilization	43.6%	12.3%	41.7%	43.9%

Table II. Impact Of Bit Precision On CPU & FPGA Efficiency

	32-bits	16bits	8-bits	4-bits	2-bits	1-bits
Dim (D)	1.2K	2.1K	3.6K	5.6K	7.5K	8.8K
EDP CPU	1x	1.11x	1.83x	2.24x	3.1x	4.02x
EDP FPGA	12.90x	11.39x	6.91x	5.64x	4.08x	4.19x

4. HDC-BASED REINFORCEMENT LEARNING

A. RL task definition

The primary goal of Reinforcement Learning (RL) is training an agent's capabilities of maximizing rewards when interacting with its environment [14, 16, 18]. Based on whether an agent uses a policy to select its action for each time step t, we can divide the RL algorithm into policy-based RL, such as Proximal Policy Optimization (PPO) [18], and off-policy RL, such as Deep Q-Learning Network (DQN) [14]. Take DQN as an example, at each time step t, an agent receives its state s_t from the environment and performs an action a_t to the environment. The agent maintains a Q function to select the action based on its current state s_t. After conducting a_t to the environment,

the agent will receive a reward r_t as feedback and transfer into a new state s_{t+1}. The agent will repeat these interactions with the environment and try to maximize the cumulative reward $R_t = \sum_{i=t}^{T} \gamma^{i-t} * r_t$, where T is the episode's total time, or trajectory length, and $\gamma \in (0,1]$ is the time step discount factor.

B. Hyperdimensional regression algorithm

Regression is a kind of supervised learning that is used to predict continuous function values given its independent variables. It has been widely used in data analysis and is also an indispensable component in RL algorithms. To find the causal dependencies between the variables, regression techniques generally 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.

On the other hand, HDC can serve as a lightweight regressor by mapping the original function space to the hyperspace. In HDC-based regression [25], by using the kernel-based encoder defined in, we can construct a hyperdimensional representation of function, similar to with the mapping Z_D: $\vec{R} = \sum_k \alpha_k Z_D(x_k)$. We refer to this

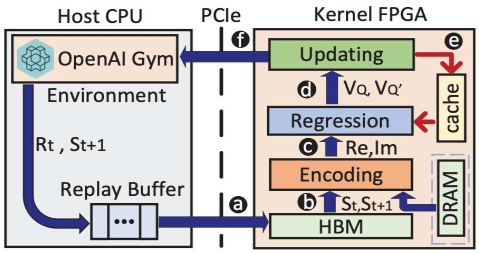


Fig. 3. Hyperdimensional Q-Learning overview on CPU-FPGA Platform.

mapping Z_D as an HDC encoder that outputs encoded hypervectors $Z_D(x)$. The representation $\vec{R}$ shows that we can approximate the function through a weighted sum of encoded training samples, which makes itself also a hypervector. In addition, we refer to $\vec{R}$ as the model hypervector, and the inference is simply the inner product between the model and encoded hypervector: $f(x) = \overrightarrow{R^T} Z_D(x)$. Notice that the complex conjugate is omitted because $Z_D(x)$ has only real components. To update the model hypervector $\vec{R}$, we feedback the prediction error as the weight for the corresponding encoded input. Assume a true value V_{true} and a predicted value $V_{pred} = \overrightarrow{R^T} Z_D(x_k)$, the update step for the model is: $\vec{R} = \vec{R} + (V_{true} - V_{pred}) Z_D(x_k)$. This update process is essentially tuning the parameter α_k for a particular training sample x_k through the hypervector elementwise add/subtract operation, which is highly parallelizable and lightweight.

C. Efficient RL via HDC

Regression is heavily used in RL to evaluate a certain state, in terms of how much total reward can be acquired in expectation if starting from this state. To begin with, in the hyperdimensional Q-learning algorithm [14], we combine a random exploration strategy with the greedy policy, i.e., ϵ-decay policy. Assuming the action space $\mathcal{A}$ and time step t:

$$A_t = \begin{cases} random\ action\ A \in \mathcal{A}, with\ probability\ \epsilon \\ argmax_{A \in \mathcal{A}} Q(S_t, A), with\ probability\ 1-\epsilon \end{cases} \quad (6)$$

The probability of selecting random actions will gradually drop after the agent explores and learns for several episodes. With a rate of changing ϵ-decay less than 1; this ensures agents start to trust their learned model gradually. $Q(S_t, A)$ is a hyperdimensional regression model that returns approximated Q-values for input action-state pairs. Once an action A_t is chosen, the agent interacts with the environment. We then obtain the new state S_{t+1} for the agent and the feedback reward R_t from the environment. At the next time step $t+1$, we select another action A_{t+1}. This chain of actions and feedback rewards form a trajectory or an episode until some termination conditions are met. To train an RL algorithm, these episodes or past experiences are usually saved to local memory as training samples. More specifically, we save a tuple of four elements for each step in the experience replay buffer: (S_t, A_t, R_t, S_{t+1}).

The model is trained at the end of each time step after saving current information to the replay buffer. We then apply a strategy called experience replay; we will sample a one-step experience tuple from past trajectories in the replay buffer to train our HDC regression model. To update the model, we first encode the input state S_t to the hypervector $\overrightarrow{S_t}$ and the predicted value q_{t_pred} is simply calculated as the dot product. Since most RL tasks can be viewed as a Markov Decision Process (MDP), the Bellman equation gives a recursive expression for the Q-value at step t. To learn the HDC regression model that represents the optimal Q-function, we use the Bellman optimality equation as shown below:

$$q_{t\_true} = R_t + \gamma max_A Q'(S_{t+1}, A) \quad (7)$$

Here we use a delayed model Q' that gets updated periodically using parameters in Q, known as the Double Q-Learning. We also include a reward decay term γ that adjusts the effect of future rewards on the current step Q-value. Finally, we update the model corresponding to the action taken, using the error $q_{t\_true} - q_{t\_pred}$ and the encoded state hypervector, with the learning rate β:

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.

Table III. Comparison Table with previous RL Acceleration works

	ASPLOS'19[40]	FCCM'20[41]	ICCAD'20[42]	IPDPSW'21[43]	DAC'21[44]	DARL1
Platform	Xilinx VCU1525	Alveo U200	ASIC	PYNQ-Z1	Alveo U50	Alveo U280
Clock	180MHz	285MHz	800MHz	100MHz	164MHz	171MHz
Algorithm	A3C	PPO	A3C	DQN	DDPG	HDQL
Task Env	Discrete	Continuous	both	Discrete	Continuous	Discrete
Precision	Floating 32-bit	Floating 32-bit	-	Fixed 32-bit	Fixed 32, 16-bit	Fixed 32-bit
DSP	2348	3744	-	4	2302	17
Model Size	2592.0 KB	229.6 KB	-	-	514.4 KB	64 KB
Throughput	12849.1 IPS	6823.2 IPS	+	+	38779.8 IPS	36597.1 IPS
Energy Efficiency	141.7 IPS/W	-	+	+	2638.0 IPS/W	5256.3 IPS/W

$$\overrightarrow{M_{A_{t+1}}} = \overrightarrow{M_{A_t}} + \beta\left(q_{t\_true} - q_{t\_pred}\right)\overrightarrow{S_t} \quad (8)$$

D. Hardware Acceleration Design

Fig. 3 presents the CPU-FPGA architecture of hyperdimensional reinforcement learning acceleration framework [24]. The agent's interaction with the environment is run on a CPU, and a replay buffer is maintained on the same host CPU. To accelerate high-dimensional vectors (called hypervectors) operations during training and inference, the host CPU will offload corresponding state, action, and reward data to the FPGA kernel via PCIe communications as shown in Fig. 3 (a). After finishing the hypervector computation, the kernel FPGA will return to training or inferring results back to the host CPU. The hypervector computation on the FPGA includes three layers: the encoding layer (Encoding), the regression layer (Regression), and the model updating layer (Updating). The FPGA kernel reads the input data, such as the state, action mask, and reward, via the AXI interface from DRAM or HBM. The quantization precision that we chose here is a fixed point-32 bit. The original state vector is encoded into a HDC vector inside the encoding layer. The kernel function that we selected for this layer's encoding is an exponential function. The encoded HDC vectors will then be loaded into the regression layer (Fig 3. (c)). The generated Q value will load into the updating layer (Fig 3. (d)). Two operations occur inside this layer. The first is the selection of the optimal action index and relaying it back to the host CPU (Fig 3. (f)). The second is to generate the model update value and store it in the on-chip cache (Fig 3. (e)).

In Table III, we compare our acceleration of HDC-based RL algorithm with previous RL acceleration works. We mainly focus on the comparison of DSP utilization, model size, throughput, and energy efficiency.

Table IV. Resource Utilization and Performance on Alveo U280

	CartPole	Lunar Lander
LUT	73.1K	117.4K
BRAM	276	546
UltraRAM	79	143
FF	38047	42508
DSP	17	17
f (MHz)	171 MHz	171 MHz
L (cycle)	417	624

E. Results

We implemented hyperdimensional reinforcement learning (HDRL) accelerator on Xilinx Alveo U280. The host CPU is Intel Xeon 6226. The reinforcement leaning environment includes OpenAI Gym CartPole and LunarLander. Table IV presents the resource utilization. In Table IV, we compare our HDRL FPGA accelerator with state-of-the-art deep reinforcement leaning FPGA accelerator on both learning throughput and energy efficiency.

5. HDC Graph Reasoning

A. Problem Definition

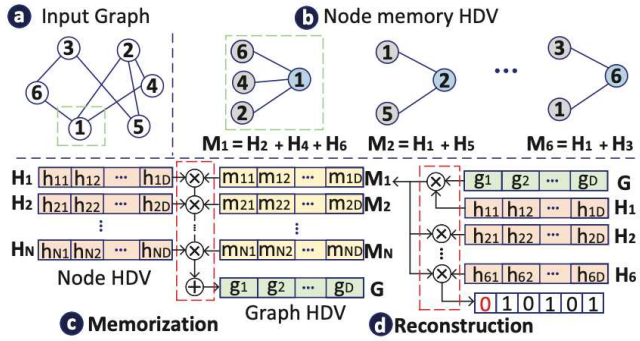


Fig. 4. Hyperdimensional graph reasoning (HGR) example.

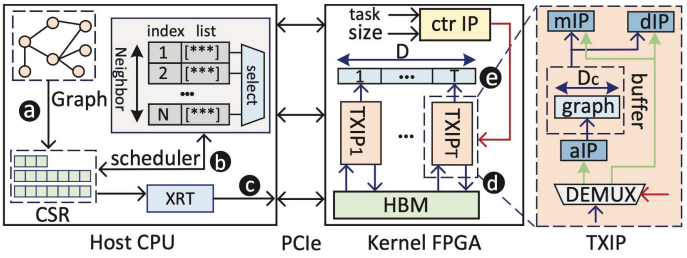


Fig. 5. Hyperdimensional Graph Reasoning Acceleration on CPU-FPGA Computing Platform.

Memorization and reconstruction are essential functionalities that enable machine learning algorithms to provide a high quality of learning and reasoning for each prediction [4, 29, 36]. Fig. 4 shows the general hyperdimensional graph reasoning (HGR) procedure. HGR supports two high-level reasoning tasks: graph memorization (memorization) and graph reconstruction (reconstruction). Graph memorization is the process of compressing the information of a graph into a single hypervector. Graph reconstruction aims to rebuild the relations between entities based on the previously done memorization.

B. HDC Model Design

The memorization process (Fig. 4 (a)) includes two steps: the node memory hypervector generation, and node memory bundling. The memory hypervector is generated by aggregating each node's neighbors' feature hypervectors. Fig. 4 (b) provides an example of node memory hypervectors generation based on the graph shown in Fig. 4 (a). The memory bundling process consists of binding each node's hypervector with its memory hypervector and bundling the results across all nodes (Fig. 4 (c)). Fig. 4 (d) gives an example of graph node memory reconstruction. To determine each node's neighbor nodes, first we need to reconstruct each node memory hypervector. The second is to remove the noise of the reconstructed memory hypervector.

C. Hardware Acceleration Design

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.

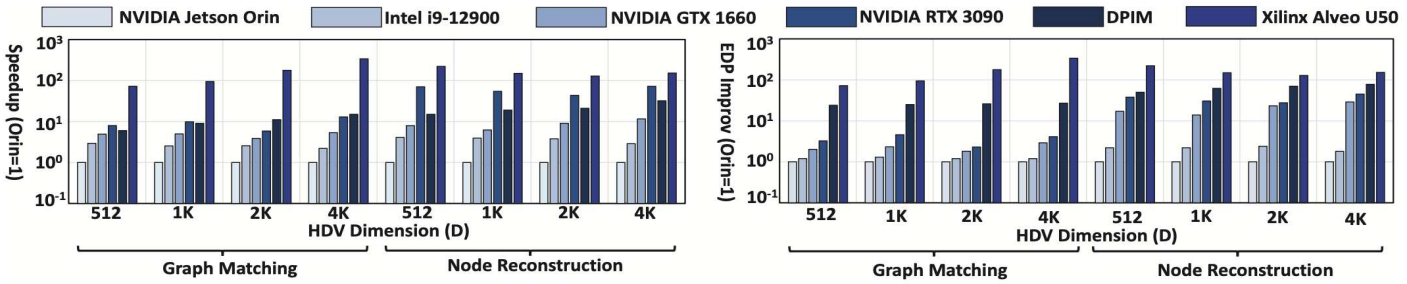


Fig. 6. Hyperdimensional graph reasoning (HGR) acceleration performance on different platforms.

Figure 5 is an overview of the CPU-FPGA heterogeneous platform. We first convert the graph representation from adjacency matrix format into compressed sparse row (CSR) format. Although the CSR format successfully diminishes the matrix's sparsity, it also incurs the computing workload imbalance problem as discussed by previous works [20], [25]. To fix this issue, we design an out-of-order (OoO) style, density-aware scheduler running on the CPU. The scheduler will offload actual hyperdimensional computing (HDC) activities on kernel FPGA. Here suppose the hypervector dimension is D. To parallelize the matrix multiplication (MM), we split each graph node's hypervector into T chunks. Each chunk's dimension $Dc = \frac{D}{T}$. As shown in Fig 5, those T vectors will be first loaded into T high bandwidth memory (HBM)'s channels and then accessed by T independent transaction IP (TXIP). Inside each TXIP, there is one aggregator IP (aIP) and one decoder IP (dIP). The aggregator IP will conduct memorization computing, and the decoder IP will conduct reconstruction computing. After each TXIP finishes its computing activities, we concatenate each channel's chunk and generate the result hypervector.

Table V. FPGA Resource Utilization on Xilinx Alveo U50 with the Frequency is 200MHz and the Power Consumption is 29.8W

	LUT	FF	BRAM	UltraRAM	DSP
aIP	244.4K	101.7K	128	0	0
dIP	268.3K	122.8K	0	64	2048
HBM	4320	3496	16	0	9
Other	72.1K	80.6K	94	0	0
Total	589.1K	308.7K	238	64	2048

D. Results

We implemented HGR accelerator on Xilinx Alveo U50. Table V presents resource utilization. Here we try to implement aggregator IP and decoder IP on the same FPGA but the implementation of them can be separated. In Fig. 6 We compare HDR FPGA accelerator with multiple different hardware platforms including NVIDIA GTX 1080, RTX 3090, Jetson Orin, and previous PIM accelerator.

6. HDC GENOMICS

A. Problem Definition

The inherent sequential processes of genome matching, which can be computationally intensive and slow. The process of predicting the possible DNA/RNA sequence that a specific protein has originated from is called back-translation. Aligning the back-translated RNA sequence against the database locates the most similar sequences used to predict the functionality of the unknown protein sequence. Proteins are made up of one or more chains of 20 common amino acids. An unknown protein can be characterized when its sequence shares significant similarity with a protein with known characteristic.

B. HDC Model Design

Fig. 7 presents the overview of BioHD [3] sequence search in the high- dimensional space. The first step of BioHD is to map the genome sequence into a high-dimensional space. BioHD assigns a hypervector corresponding to each base alphabet Σ={A, C, G, U} in for DNA and Σ={A, C, G, U} for RNA. The encoding module depends on the data type and the genomics task. In terms of protein data, BioHD assigns a hypervector representing each RNA base and then combines them to create a hypervector representing each amino acid. The amino acids' hypervectors are combined by mapping each protein sequence into a high-dimensional space. BioHD aggregates all encoded protein sequences to generate a reference genome, called HDC Library. An HDC library consists of several reference hypervectors, where each hypervector memorizes thousands of genome sequences in high-dimensional space. Similar to the human memorization that requires practice, BioHD iteratively checks the correctness of memorized information in each library hypervector to find the most refined hypervectors. During the sequence searching, BioHD uses the same encoding to map a query sequence into a hypervector. We perform a similarity computation between a query and each reference hypervector. By searching for an exact or approximate match, BioHD identifies a query's closeness with thousands of memorized patterns stored in each HDC library hypervector.

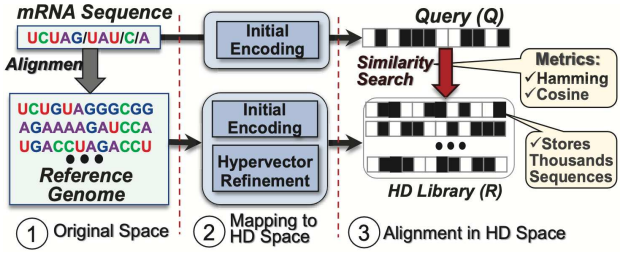


Fig. 7. Hyperdimensional genomic sequence matching.

C. Hardware Acceleration Design

The hardware design of work [3] is centered around a Processing In-Memory (PIM) architecture. This architecture is designed to be compatible with existing crossbar memory and supports all essential BioHD operations natively in memory with minimal modification on the array. Fig. 8 shows the architecture design of PIM. Fig. 8 (A) shows an overview of the PIM architecture consisting of 128 tiles. Each tile consists of 128 crossbar memory blocks. Due to the existing challenges of crossbar memory, each memory block is assumed to have a size of 1Kx1K. BioHD consists of two types of memory blocks: encoding and distance computing. Both blocks are the same conventional crossbar array; they are organized in each tile to enable fast and parallel sequence searching. As is shown Fig. 8 (B), the block sense amplifiers are low-precision ADCs that are shared among several memory blocks. Unlike the existing analog-based PIM, crossbar memory takes up the majority of our architecture, and ADCs only take up a tiny area. Fig. 8 (C) shows the first step of the encoding that assigns a codebook to each amino acid. Fig. 8 (D) shows that for a block with 1k-columns, each block stores 32 permutations of the acid hypervectors. Each 32-columns stores all possible hypervectors that a protein in a specific position can take. These hypervectors can be directly addressed using our codebooks. Fig. 8 (E) shows a row-parallel Hamming computing between a query and all reference hypervectors stored in the memory. This work uses 32-bits windows to ensure 5-bit ADC precision. To limit the cost of ADC blocks, we share ADCs among multiple distance computing blocks. We use sample & hold (S+H) circuit to record the matching line discharging current of each block and use time multiplexing to share ADC among 128 memory blocks. Fig. 8 (F) shows row-parallel dot product operation between a query and stored reference hypervectors. Fig. 8 (G) shows the whole system's working pipeline.

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.

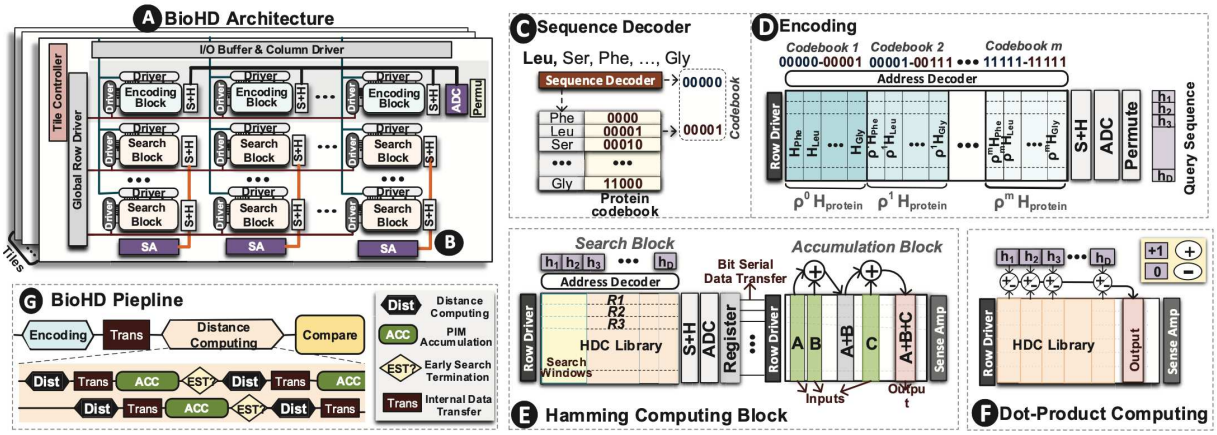


Fig. 8. Overview of PIM-based BioHD architecture along with encoding and distance computing blocks.

D. Results

Table VI. Detailed Configurations of BioHD

BioHD PIM Parameter				
Component	***Param***	***Spec***	***Area***	***Power***
Crossbar Array	size	1Mb (1kx1k)	3136um2	6.14mW
Sense Amp	number	1k	49.2um2	0.09mW
Memory Block	number	1	3185.2um2	6.23mW
Hamming Computing (HAM)				
Tile Memory	number	128 blocks	0.40mm2	0.78W
ADC	resolution	5-bits, 1.75GS/s	0.16mm2	0.14W
S+H	number	128	857.6um2	14.97mW
Interconnect	size	128x1k	0.01mm2	31.04mW
Controller	number	1k/row	146.5um2	118.9mW
HAM-tile	size	128Gb	0.52mm2	1.07W
Dot Product Computing (DOT)				
Tile Memory	number	128 blocks	0.40mm2	0.78mW
Interconnect	size	1k/row	0.01mm2	31.04mW
Controller	number	1	146.5 mm2	118.9mW
DOT-tile	size	128Gb	0.41 mm2	0.93W
HAM Chip	number size	128 Tiles 2GB	73.52 mm2	137.81W
DOT Chip	number size	128 Tiles 2GB	53.04mm2	119.79W

For the hardware design, we use HSPICE for circuit-level simulations to measure the energy consumption and performance of all the BioHD operations in 28nm technology. We used System Verilog and Synopsys Design Compiler to implement and synthesize the BioHD controller. For parasitic, we used the same simulation setup considered by work in. The interconnects are modeled in both circuit and architecture levels. The robustness of all circuits has been verified by considering 10% process variations on the size and threshold voltage of transistors. Our PIM works with any bipolar resistive technologies, which are the most common NVMS. To have the highest similarity to commercially available 3D Xpoint, we adopt the memristor device with a VTEAM model. The memristor's model parameters are chosen to produce a switching delay of 1ns, a voltage pulse of 1V and 2V for RESET and SET operations to fit practical devices.

Table VI shows the detailed configurations of BioHD consisting of 128 tiles. Each tile has 128 crossbar blocks. BioHD has two configurations: Hamming computing that uses shared ADC blocks for distance computing, and Dot Product computing (DOT), where the distance is computed using row-parallel PIM arithmetic. In DOT- tile, the crossbar memory takes most of the area and power consumption, while in HAM-tile, ADCs are taking 28% and 15% of total area and

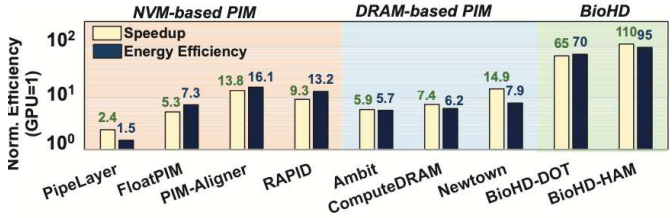


Fig. 9. Hyperdimensional genomic sequence PIM comparison with state of the art.

power consumption. Each HAM-tile (DOT-tile) takes 0.57$mm2$ (0.41$mm2$) area and consumes 1.07W (0.93W) power. The total HAM-chip (DOT-chip) area and average power consumption are 73.52$mm2$ and 137.81W (53.04$mm2$ and 119.79W), respectively [30]. All our evaluations are performed when BioHD provides the same area in both configurations. Note that our HAM chip can be configured to perform both Hamming distance and dot product similarity, while DOT Chip is an optimized version that just supports dot product similarity.

Fig. 9 compares BioHD efficiency with other PIM accelerators. All PIMs have the same area as BioHD in the 1-chip configuration. The efficiency values are reported compared to GPU. We compute the efficiency of PIM accelerator using our cycle-accuracy simulator. The results are validated with the performance and efficiency reported on each original paper. PipeLayer [6] and FloatPIM [7] are neural network accelerators, but their operations can be used to accelerate the sequence searching algorithm. Our evaluation shows that BioHD provides significant efficiency improvement compared to PIM architectures. This efficiency comes from: (i) BioHD capability in revisiting alignment using HDC with hardware-friendly operations, (ii) BioHD PIM architecture supporting highly parallel essential operations, and (iii) data flow in BioHD that eliminates in- ternal data movement.

Fig. 9 also compares BioHD efficiency with DRAM-based accelerators: Ambit [11], ComputeDRAM [17], and Newton [13]. DRAM-based PIMs are suitable to accelerate existing alignment algorithms that rely on extensively parallel bitwise and arithmetic computation. In contrast, these accelerators do not support associative search, which is the key functionality of BioHD computation. This makes DRAM-based solution ineffective for BioHD acceleration. Our evaluation shows that, in the same area, BioHD provides 7.3× and 12.0× (14.8× and 15.3×) faster and higher energy efficiency com- pared to Newtown (ComputeDRAM).

7. Conclusion

In this paper, we summarize prior hardware-algorithm codesigns of HDC accelerator targeting different machine leaning applications.

Acknowledgement:

This work was supported in part by DARPA Young Faculty Award, National Science Foundation #2127780, #2319198, #2321840 and

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.

#2312517 , Semiconductor Research Corporation (SRC), 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 generous gifts from Xilinx and Cisco.

References:

[1] M. Imani *et al*., "Neural computation for robust and holographic face detection." in DAC. ACM/IEEE, 2022.
[2] P. Poduval *et al*. "Adaptive neural recovery for highly robust brain-like representation." in DAC. ACM/IEEE, 2022.
[3] Z. Zou *et al.,* "BioHD: an efficient genome sequence search platform using HyperDimensional memorization." In Proceedings of the 49th Annual International Symposium on Computer Architecture.
[4] P. Poduval *et al*., "Graphd: Graph-based hyperdimensional memorization for brain-like cognitive learning." Frontiers in Neuroscience 16 (2022): 757125.
[5] P. Poduval *et al*., "Cognitive correlative encoding for genome sequence matching in hyperdimensional system." in DAC. ACM/IEEE, 2021.
[6] L.Song, et al,"Pipelayer:Apipelinedreram-basedacceler- ator for deep learning," HPCA, 2017.
[7] M. Imani, et al, "Floatpim: In-memory acceleration of deep neural network training with high precision," in ISCA. ACM,2019.
[8] H. Barkam *et al*., "Comprehensive Analysis of Hyperdimensional Computing Against Gradient Based Attacks." in DATE. IEEE, 2023.
[9] H. Barkam *et al*., "HDGIM: Hyperdimensional Genome Sequence Matching on Unreliable highly scaled FeFET," in DATE. IEEE, 2023.
[10] Z. Zou *et al.,* "EventHD: Robust and efficient hyperdimensional learning with neuromorphic sensor," Frontiers in Neuroscience, 2022.
[11] S. Angizi, et al, "Pim-aligner: a processing-in-mram platform for biological sequence alignment," DATE. IEEE, 2020.
[12] Z. Zou *et al.*, "Scalable Edge-Based Hyperdimensional Learning System with Brain-Like Neural Adaptation," SC21: International Conference for High Performance Computing, Networking, Storage and Analysis, 2021.
[13] M. Hee, et al, "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning. in MICRO, IEEE, 2020.
[14] Y. Ni *et al*., "Efficient Off-Policy Reinforcement Learning via Brain-Inspired Computing." in GLSVLSI. ACM, 2023.
[15] Y. Ni *et al*., "Neurally-Inspired Hyperdimensional Classification for Efficient and Robust Biosignal Processing." in ICCAD. ACM, 2022.
[16] M. Issa *et al*., "Hyperdimensional Hybrid Learning on End-Edge-Cloud Networks," in ICCD. IEEE, 2022.
[17] F. Gao, et al, "Computedram: In-memory compute using off-the-shelf drams," in MICRO. ACM/IEEE, 2019.
[18] Y. Ni *et al.,* "HDPG: hyperdimensional policy-based reinforcement learning for continuous control," In Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22), 2022.
[19] S. Shahhosseini *et al*., "Flexible and Personalized Learning for Wearable Health Applications using HyperDimensional Computing." Proceedings of the Great Lakes Symposium on VLSI 2022. 2022.
[20] Y. Ni *et al*., "Algorithm-hardware co-design for efficient brain-inspired hyperdimensional learning on edge." DATE. IEEE, 2022.
[21] Y. Ni *et al.,* "Online Performance and Power Prediction for Edge TPU via Comprehensive Characterization," DATE. IEEE, 2022.
[22] H. Lee *et al*., "Comprehensive Integration of Hyperdimensional Computing with Deep Learning towards Neuro-Symbolic AI," 2023 60th ACM/IEEE Design Automation Conference (DAC), 2023.
[23] A. Hernández-Cano *et al.,* "Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system." DATE. IEEE, 2021.
[24] H. Chen *et al*. "DARL: Distributed Reconfigurable Accelerator for Hyperdimensional Reinforcement Learning." Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design. 2022.
[25] H. Chen *et al.,* "Full Stack Parallel Online Hyperdimensional Regression on FPGA," 2022 IEEE 40th International Conference on Computer Design (ICCD), 2022.
[26] H. Chen *et al*., "Density-Aware Parallel Hyperdimensional Genome Sequence Matching," 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2022.
[27] A. Kazemi *et al*., "Achieving software-equivalent accuracy for hyperdimensional computing with ferroelectric-based in-memory computing." Scientific reports 12.1 (2022): 19201.
[28] M. Imani *et al.,* "Dual: Acceleration of clustering algorithms using digital-based processing in-memory." in MICRO. IEEE, 2020.
[29] H. Chen *et al.,* "HyperGRAF: Hyperdimensional Graph-Based Reasoning Acceleration on FPGA," 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL), 2023.
[30] X. Yin *et al*., "An Ultracompact Single‐Ferroelectric Field‐Effect Transistor Binary and Multibit Associative Search Engine." Advanced Intelligent Systems (2023).
[31] S. Shou *et al*., "SEE-MCAM: Scalable Multi-bit FeFET Content Addressable Memories for Energy Efficient Associative Search." Proceedings of the 42st IEEE/ACM International Conference on Computer-Aided Design. 2023.
[32] C. Liu *et al.,* "Cosime: Fefet based associative memory for in-memory cosine similarity search." in ICCAD. IEEE/ACM, 2022.
[33] Kanerva, Pentti. "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors." Cognitive computation 1 (2009): 139-159.
[34] M. Hersche *et al.,* "Exploring embedding methods in binary hyperdimensional computing: A case study for motor-imagery based brain-computer interfaces." arXiv preprint arXiv:1812.05705 (2018).
[35] A. Manabat *et al.,* "Performance analysis of hyperdimensional computing for character recognition." 2019 International Symposium on Multimedia and Communication Technology (ISMAC). IEEE, 2019.
[36] Kang, Jaeyoung, et al. "RelHD: A Graph-based Learning on FeFET with Hyperdimensional Computing." ICCD. IEEE, 2022.
[37] M. Imani *et al.,* "Voicehd: Hyperdimensional computing for efficient speech recognition." 2017 IEEE international conference on rebooting computing (ICRC). IEEE, 2017.
[38] M. Imani *et al.,* "Hdcluster: An accurate clustering using brain-inspired high-dimensional computing." 2019 Design, Automation & Test in Europe Conference \& Exhibition (DATE). IEEE, 2019.
[39] A. Rahimi *et al.,* "Random features for large-scale kernel machines." Advances in neural information processing systems 20 (2007).
[40] Cho, Hyungmin, et al. "Fa3c: Fpga-accelerated deep reinforcement learning." ASPLOS. ACM, 2019.
[41] Meng, Yuan, et al. "Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms." FCCM. IEEE, 2020.
[42] Wang, Ying, et al. "A many-core accelerator design for on-chip deep reinforcement learning." ICCAD. ACM, 2020.
[43] Watanabe, et al. "An FPGA-based on-device reinforcement learning approach using online sequential learning." IPDPSW. IEEE, 2021.
[44] Yang, Je, et al. "Fixar: A fixed-point deep reinforcement learning platform with quantization-aware training and adaptive parallelism." DAC. IEEE, 2021

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on August 07,2024 at 02:50:46 UTC from IEEE Xplore. Restrictions apply.