2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)
Phoenix, Arizona, USA, August 6-9, 2023

Hardware-Optimized Hyperdimensional Computing for Real-Time Learning

Hanning Chen, Hamza Errahmouni Barkam, Mohsen Imani
University of California, Irvine, Irvine, CA 92697, USA
Email: {hanningc, herrahmo, m.imani}@uci.edu

2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS) | 979-8-3503-0210-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/MWSCAS57524.2023.10406123

***Abstract*—Reinforcement learning presents a promising approach to bolster cybersecurity through the development of intelligent agents capable of learning from their environment and adapting to new threats. In the field of cybersecurity, reinforcement learning has various applications, including intrusion detection, malware classification, and vulnerability analysis. However, current reinforcement learning algorithms such as Deep Q-Learning rely on deep neural networks, which entail high computational costs and unsuitability for deployment on edge devices. To overcome this challenge, we proposed two solutions for efficient reinforcement learning on edge devices.**

The first solution is a Hyperdimensional Reinforcement Learning algorithm inspired by the brain's properties that facilitate robust and real-time learning using a lightweight brain-inspired model to learn an optimal policy in an unknown environment. Next, we propose a heterogeneous CPU-FPGA platform that maximizes the computing capabilities of FPGAs by applying hardware optimizations for hyperdimensional computing's critical operations. Our platform achieves faster and higher energy efficiency than state-of-the-art reinforcement learning accelerators while maintaining the same or better quality of learning. Additionally, we enhance the RL model's learning capabilities, such as learning throughput, energy efficiency, and robustness. Our proposed solutions offer efficient and scalable alternatives for reinforcement learning on edge devices, making it possible to support online and real-time learning with minimal memory capacity.

***Index Terms*—cybersecurity, reinforcement learning, hyperdimensional computing, brain-inspired learning, q-learning**

I. Introduction

Reinforcement learning (RL) is widely used in different domains, including cybersecurity, to tackle complex and dynamic decision-making problems that traditional rule-based and signature-based methods are not suitable for. RL algorithms can be trained to identify and respond to attacks, optimize security policies, and even generate new attack scenarios. Although the use of RL in cybersecurity is still in its early stages, it has shown promising results and has become an active area of research.

Compared to traditional machine learning techniques, RL does not require large amounts of labeled data for training. Instead, it involves defining an environment that simulates the task the model needs to learn, where the agent has no prior knowledge of the environment. In RL, there are two primary approaches to achieving this: policy-based learning and value-based learning. Q-Learning is an algorithm that computes a value for each state in the environment to measure the expectation of attaining future rewards, but in large state and action spaces, a deep neural network is required to compute these values, which can be computationally expensive during backpropagation.

Hyper-Dimensional Computing (HDC) has recently emerged as a highly efficient and robust machine learning model. HDC encodes objects with high-dimensional data representations called hypervectors, which store thousands of elements and incorporate memory functions and vector operations, making it computationally tractable and mathematically rigorous. HDC can learn from a few shots of training data and is analogous to human-like learning and reasoning capabilities.

Our recent work uses this HDC model as an alternative to Deep Q-Learning for reinforcement learning tasks [1], [2]. This HDC-based RL model is able to achieve comparable learning outcomes to a Deep Q-Learning Model with the advantage of being computationally more efficient while also producing significantly higher learning quality, i.e., faster learning and convergence. This advantage is particularly important in edge computing since HDC algorithms require far less computational power due to the simple arithmetic of these algorithms compared to the complexity of Neural Networks [3]–[9]. Despite the success, HDC-based RL still lacks parallelism and requires a large number of resources on traditional cores [1]. Previous works already show that HDC-related operations, such as hypervector multiplication, have a long execution time on the CPU [10]–[15]. Domain-specific accelerators targeting HDC model have shown large sucess by previous works [12], [13], [16]–[19]. Furthermore, many domain-specific accelerators [20]–[24] have achieved great acceleration results of RL algorithms.

We developed an architecture and algorithm co-optimization to maximize an RL agent's learning throughput by realizing the best use of FPGA's resource utilization. We evaluate the effectiveness of our approach on classic OpenAI Gym [25] tasks. Our results show that the CPU-FPGA platform provides on average 20× speedup compared to current state-of-the-art hyperdimensional Q-Learning methods [1] that run on Intel Xeon 6226 CPU. On the Xilinx Alveo U280 accelerator card platform, drawing less than 20 watts, our accelerator shows 14× speedup over state-of-the-art reinforcement learning FPGA acceleration work using only a 73.1K look-up table for the single-agent accelerator, making it suitable for deployment on an edge device. Our platform shows flexibility on different FPGA platforms with different resources condition.

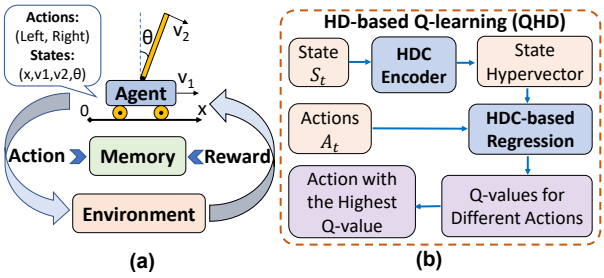

Fig. 1. Q-Learning Overview

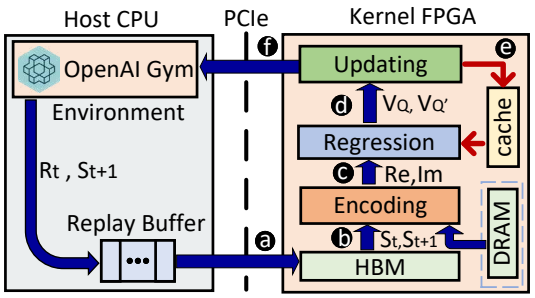

Fig. 2. HDQL acceleration overview on CPU-FPGA Platform.

II. HDC-based Q-Learning

As shown in Figure 1, the primary goal of Reinforcement Learning (RL) is training an agent's capabilities to find the reward-maximizing behavior when interacting with its environment [26]. 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), and off-policy RL, such as Q-Learning [27]. This work focuses on off-policy Q-Learning due to its fast convergence speed. Figure 1 presents a general Q-Learning procedure. 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.

The Q function is one of the most critical components of every Q-Learning algorithm. All other optimization techniques, including experience replay and the use of a target network, rely on this Q function. Traditionally, the Q function is represented as a table called the Q-table [27]. Such Q-table-based Q-Learning algorithms are referred to as Tabular Q-Learning. Tabular Q-Learning is simple but has difficulty scaling because when a task's complexity increases, the size of the Q-table will also increase, amassing a burden to memory access.

Today, most researchers use Neural Networks (NN) to implement the Q function to handle more complicated tasks [28]. These neural network-based Q-Learning algorithms are called Deep Q Learning (DQN). Although DQNs can handle complex tasks, the computing resources necessary to train these models is immense, making deploying DQN models on edge devices extremely difficult. The loss gradient calculation and backpropagation also limit the DQN model's performance improvement. To overcome these issues, our most recent work [1] uses hyperdimensional computing(HDC) to replace neural networks in the Q-Learning algorithm. It shows that HDC-based Q-Learning (HDQL) can achieve faster learning speed than DQN. Also, [1] shows that HDQL can achieve high learning rewards with limited memory access.

III. Acceleration Platform

A. Framework Overview

Figure 2 shows our designed platform's top architecture. The agent interacts with the environment 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 Figure 2 ⓐ. After the hypervector computation, the kernel FPGA will return to training or inferring results 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 ($\vec{s_t}$) is encoded into an HDC vector inside the encoding layer. The kernel function that we selected for this layer's encoding is an exponential function($f(x) = e^{jx}$), which means each element of the encoded hypervector is a complex number. Here, we call this hypervector a **complex hypervector**. To simplify the on-chip computation, we will divide this complex hypervector into the real ($\vec{Re}$) and imaginary ($\vec{Im}$) parts based on Euler's formula. The encoded HDC vectors will then be loaded into the regression layer (ⓒ). Two regression models: **Q** and **Q'**, are maintained inside this layer to conduct double estimation. After the regression layer, the generated function values: V_Q and $V_{Q'}$ will be loaded into the updating layer (ⓓ). Two operations occur inside this layer. The first is the selection of the optimal action index and relaying it back to the host CPU(ⓕ). The second is to generate the model update value and store it in the on-chip cache (ⓔ).

B. Encoding Layer Architecture

The state vector($\vec{s}$) is passed from the off-chip DRAM or on-chip HBM into the encoding layer. As is shown in Figure 3a ❶, each element of $\vec{s}$ will multiply with its corresponding position hypervector. Suppose the dimension of the state vector and position hypervector are N and D:

these N position hypervectors will be stored on-chip as a position hypervector matrix (**P**). **P**'s dimension will be $N \times D$. In the process, ❶, each row vector of matrix **P** will be multiplied with its corresponding state vector's element. In process ❷, the reduction operation will be applied over matrix **P'** row direction to generate hypervector $\vec{E}$. After the reduction operation, the generated $\vec{E}$ will be encoded using a kernel function. The kernel function that we select in this paper is the exponential function. A new hypervector $\vec{E'}$ will be generated after applying this kernel function to every element of $\vec{E}$.

Next, we elaborate on the implementation of our exponential encoder IP on FPGA. Inside each exp encoder IP, there are Sine and one Cosine encoder IP. Many efficient methods exist for the FPGA hardware design to implement triangle functions, such as Taylor Expansion or using the Vitis HLS math function. We use the **triangle codebook** method to save on-chip resource utilization and reduce computing latency. We use BRAM to store the pre-computed Sine and Cosine values. We refer to this BRAM with the stored Sine and Cosine values as a codebook. The original fixed-point number will be treated as an address to access those on-chip BRAMs during the Sine/Cosine computing process. The benefits of using this codebook include reducing resource utilization, especially LUT and DSP, and saving calculation time.

C. Regression Layer Architecture

Inside the regression layer, the encoded state hypervectors $\vec{Re}$ and $\vec{Im}$ will have double regression performed on them. Here, double regression indicates that there are two action models inside this layer. We define them as **Q** and **Q'**. The dimensions of the Q and Q' hypervector matrices are $A \times D$. Here, A is the task's action space, and D is the hypervector's dimensionality. Each row hypervector of the **Q** and **Q**′ matrices represent the corresponding action's Q function. Like double deep Q-Learning, model Q' is a delayed model which will be periodically updated using parameters in model Q. The benefit of maintaining two action models is to avoid maximization bias by disentangling the updates from biased estimates. Specifically, model Q is used for learning purposes, and Q' is used for inference purposes. In contrast to traditional DQN, which separates the learning and inference processes, the two processes occur simultaneously in our platform kernel, as shown in Figure 3 processes ⓐ and ⓑ. The purpose of doing this is to reduce data transmission time and realize online learning. At the starting point of each episode, the model **Q** will flush **Q'**, as is shown in process ⓔ. The on-chip hypervector to hypervector multiplication is accelerated using a systolic array. However, before introducing the microarchitecture of systolic IP, we want to illustrate model Q's update process. The model update matrix is stored inside an update cache to reduce the FPGA accelerator's critical path. The host CPU will load the action index vector $\vec{M}$ into the kernel FPGA for each time step. The dimension of the vector $\vec{M}$ is A, and each element's value is either **True** or **False**, representing whether its corresponding action hypervector needs to be updated. The action index vector $\vec{M}$ will also be loaded into the systolic array together with the model Q, as is shown in process ⓒ and Ⓐ.

In Figure 3, we also present the systolic array microarchitecture. One of the interesting hardware design tricks here is that we resize the original hypervector from a single size D vector into $\frac{D}{M}$ size M vectors. As is shown in Figure 3, there are a total of $\frac{2*D}{M}$ systolic array IP in the regression layer. Inside each systolic array, the size **M** vector will be multiplied with a size **A** $\times$ **M** matrix. This process will happen for both real and imaginary parts, as is shown for process Ⓑ and Ⓒ. The multiplication result from the real part will then be subtracted by the imaginary part. After all $\frac{2*D}{M}$ groups of systolic array operations, two reduction IPs($\sum$) are used to reduce those $\frac{2*D}{M}$ multiplication results into two vectors: $\vec{V_Q}$ and $\vec{V_{Q'}}$. Here, $\vec{V_Q}$ is the regression result for the learning process and $\vec{V_{Q'}}$ is the regression result for the inference process. Both of these two vectors' dimensions is A.

D. Updating Layer Architecture

In Figure 4.a, the action index vector ($\vec{M}$) specified by the host CPU is loaded into the select IP to choose the appropriate element from $\vec{V_Q}$ and $\vec{V_{Q'}}$. For the inference stage, all elements of $\vec{M}$ will be False and select IP will choose the largest elements' index of $\vec{V_{Q'}}$. This index is actually the action index(a_t) at time step t, which will then be passed back to the CPU and stored in the replay buffer. For the training stage, only one element of $\vec{M}$ will be set to True. The prediction value q_{pred} and true value q_{true} will be selected from $\vec{V_Q}$ and $\vec{V_{Q'}}$ respectively based on the element's value of $\vec{M}$.

In Figure 4.b, the updated value is calculated. In Figure 4.c and Figure 4.d, V_{update} will first multiply the real and imaginary parts of the encoded state vector($\vec{Re}$ and $\vec{Im}$), and then flush the update cache based on the action index vector $\vec{M}$. Specifically, in Figure 4.d, the update action hypervector is stored inside the update cache. In the future clock cycle, when a specific action hypervector will be used in the regression layer, the corresponding update hypervector will be used to update the Q hypervector matrix. By cutting the backpropagation update process into two stages, as is shown by the Vitis HLS synthesis result, the critical path of the kernel accelerator is shortened.

IV. Evaluation

We develop a full library using two modules: (1) an optimized software implementation using Python library-supported encoding and learning phase of our reinforcement learning, and (2) hardware implementation of RL on CPU, GPU, and FPGA platforms. We used Intel Xeon 6226 at 2.9 GHz as the host CPU. We use Xilinx Alveo U280 for the kernel acceleration. We also used the Xilinx Vitis framework to conduct the communication between CPU and FPGA via PCIe. We choose the benchmark from OpenAI Gym [25], including CartPole [29] and LunarLander [30].

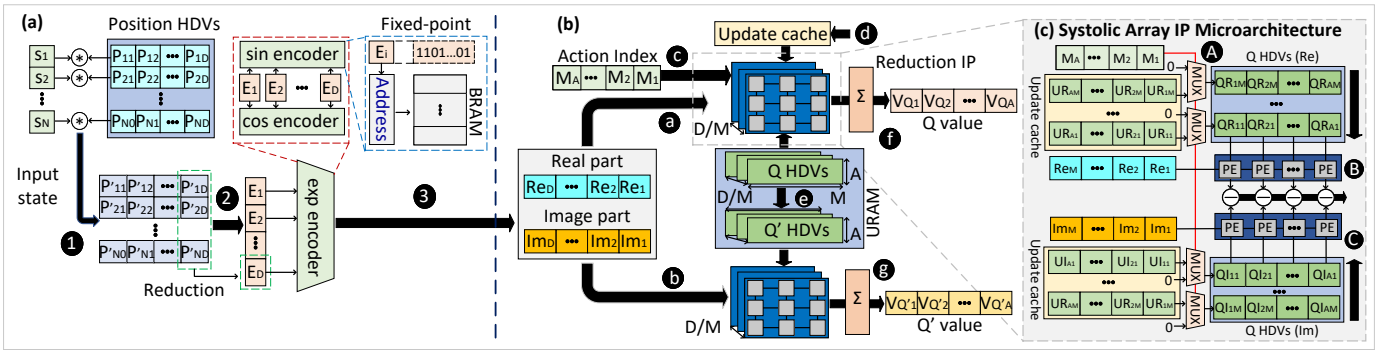

Fig. 3. (a) Encoding layer architecture design. (b) Regression layer architecture design with Systolic Array IP microarchitecture

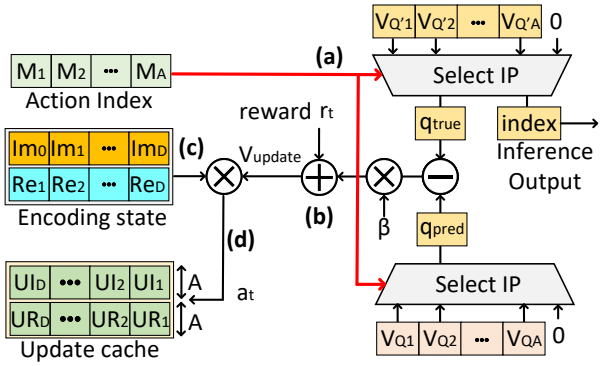

Fig. 4. Updating layer architecture design. (a)Q and Q' value selection. (b)Update value Calculation. (c)Multiply the updated value with the encoded state hypervector. (d) Store update hypervector into Update cache

TABLE I
RESOURCE UTILIZATION AND PERFORMANCE ON ALVEO U280

Environment	LUT	BRAM	URAM	FF	DSP	f (MHz)	L (cycle)
CartPole	73.1K (6%)	276 (6%)	79 (8%)	38K(1%)	17	171	417
LunarLander	117.4K (8%)	546 (12%)	143 (14%)	42.5K (1%)	17	171	421

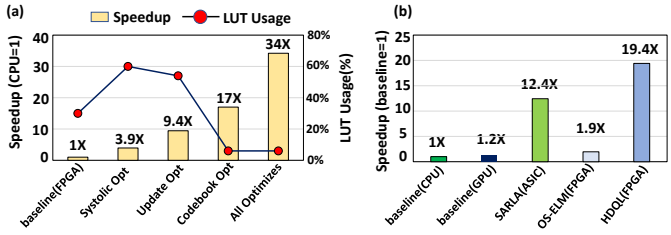

Fig. 5. Performance of our designed framework for OpenAI Gym CartPole task. (a) Impact of different optimization techniques on our platform resource utilization and speedup. (b) Different platform runtime comparison. Here we assume both SARLA [23] and OS-ELM [31]'s rewards accumulation are the same as ours.

A. FPGA Resource Utilization

Table I reports the accelerator's resource utilization of HDQL running on the Xilinx Alveo U280 platform. The synthesis and implementation tool we use is Xilinx Vitis HLS and Vivado 2021.2. Here, the hypervector dimension is 2048

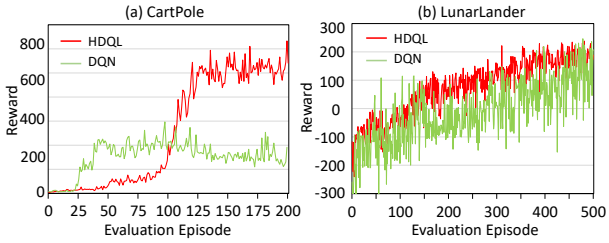

Fig. 6. The platform agents' rewards over episodes.

(2K), and each tuple's precision is a fixed-point 32-bit. The dimension of the hypervector will significantly influence the model's accuracy, which means that to achieve high rewards for learning tasks, the dimension of the hypervector cannot be shallow. We will show in the next section that a 2K fixed-point 32-bit hypervector provides enough learning rewards for both tasks. Since the two tasks, CartPole and LunarLander, have different action space and state dimensions, as shown in Table I, the resource utilization for those two tasks is different.

Table I provides detailed resource utilization for our designed accelerator with a varying number of agents. Our HDC-based Q-Learning accelerator shows very high flexibility in targeting different FPGA platforms. For example, for the accelerator with a single agent, the lookup table(LUT) usage is only **73.1K**, which indicates that our accelerator can be deployed on much smaller edge computing devices such as the Xilinx Zedboard or Zynq ZCU104 board. As a result of choosing the precision for quantizing the data to be a fixed-point 32-bit instead of a floating-point, the DSP usage in-depth decreased. Decreasing on-chip DSP usage plays an essential role in energy efficiency improvement. In Figure 5.a, we also present the three optimization's influence over the accelerator's resource utilization and speedup. Reducing the accelerator's critical path and designing a lightweight kernel encoder allows our framework to achieve high-performance learning throughput while requiring affordable resource utilization.

B. Performance: Model Accuracy and Hardware Acceleration

Figure 6 reports our HDC-based Q-Learning reward change over training episodes targeting CartPole and LunarLander tasks. Again, the dimensionality of the hypervector is 2K, and

each hypervector's tuple precision is fixed-point 32-bit. The replay buffer batch size M we selected for CartPole is eight, and for LunarLander, 16. We also test DQN [28] for the same task and report its learning results for comparison. Both single and multiple agents HDC-based Q-Learning achieve much higher rewards than DQN during the same episodes. In Figure 5.a, we also present the three optimization's influence over the accelerator's resource utilization and speedup. Reducing the accelerator's critical path and designing a lightweight kernel encoder allows for the platform to achieve high-performance learning throughput speed while requiring affordable resource utilization. We also compare our design with other platforms' acceleration results in Figure 5.b. We notice that the GPU acceleration of HDQL is not apparent. We believe this is caused by wasting a lot of time passing the Q model hypervector between CPU and GPU.

V. Conclusion

This paper presents a novel platform capable of real-time hyperdimensional reinforcement learning. Our heterogeneous CPU-FPGA platform maximizes FPGA's computing capabilities by applying several hardware optimizations to hyperdimensional computing, including hardware-friendly encoder IP, hypervector chunk fragmentation, and the delayed model update. We evaluate the effectiveness of our approach to OpenAI Gym tasks.

VI. Acknowledgement

This work was supported in part by DARPA, National Science Foundation #2127780 and #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] Y. Ni *et al.*, "Qhd: A brain-inspired hyperdimensional reinforcement learning algorithm," *arXiv preprint arXiv:2205.06978*, 2022.

[2] M. Issa *et al.*, "Hyperdimensional hybrid learning on end-edge-cloud networks," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 652–655, IEEE, 2022.

[3] Hernandez-Cane *et al.*, "Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 56–61, IEEE, 2021.

[4] Z. Zou *et al.*, "Scalable edge-based hyperdimensional learning system with brain-like neural adaptation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

[5] 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.

[6] Y. Ni *et al.*, "Hdpg: Hyperdimensional policy-based reinforcement learning for continuous control," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 1141–1146, 2022.

[7] A. Hernández-Cano, C. Zhuo, X. Yin, and M. Imani, "Reghd: Robust and efficient regression in hyper-dimensional learning system," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 7–12, IEEE, 2021.

[8] P. Poduval *et al.*, "Graphd: Graph-based hyperdimensional memorization for brain-like cognitive learning," *Frontiers in Neuroscience*, p. 5, 2022.

[9] D. Ma *et al.*, "Perfhd: Efficient vit architecture performance ranking using hyperdimensional computing," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2229–2236, 2023.

[10] M. Imani *et al.*, "Revisiting hyperdimensional learning for fpga and low-power architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 221–234, IEEE, 2021.

[11] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.

[12] H. Chen *et al.*, "Full stack parallel online hyperdimensional regression on fpga," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 517–524, IEEE, 2022.

[13] Z. Zou, H. Chen, *et al.*, "Biohd: an efficient genome sequence search platform using hyperdimensional memorization," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 656–669, 2022.

[14] Y. Ni *et al.*, "Algorithm-hardware co-design for efficient brain-inspired hyperdimensional learning on edge," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 292–297, IEEE, 2022.

[15] J. Kang *et al.*, "Openhd: A gpu-powered framework for hyperdimensional computing," *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2753–2765, 2022.

[16] H. Chen and M. Imani, "Density-aware parallel hyperdimensional genome sequence matching," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1–4, IEEE, 2022.

[17] C.-K. Liu *et al.*, "Cosime: Fefet based associative memory for in-memory cosine similarity search," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2022.

[18] M. Imani *et al.*, "Hierarchical, distributed and brain-inspired learning for internet of things systems," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2023.

[19] A. Dutta, S. Gupta, B. Khaleghi, R. Chandrasekaran, W. Xu, and T. Rosing, "Hdnn-pim: Efficient in memory design of hyperdimensional computing with feature extraction," in *Proceedings of the Great Lakes Symposium on VLSI 2022*, pp. 281–286, 2022.

[20] A. Samajdar *et al.*, "Genesys: Enabling continuous learning through neural network evolution in hardware," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 855–866, IEEE, 2018.

[21] H. Cho *et al.*, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 499–513, 2019.

[22] Y. Meng *et al.*, "Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 19–27, IEEE, 2020.

[23] Y. Wang *et al.*, "A many-core accelerator design for on-chip deep reinforcement learning," in *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–7, 2020.

[24] J. Yang *et al.*, "Fixar: A fixed-point deep reinforcement learning platform with quantization-aware training and adaptive parallelism," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 259–264, IEEE, 2021.

[25] G. Brockman *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[26] M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, and D. Hassabis, "Reinforcement learning, fast and slow," *Trends in cognitive sciences*, vol. 23, no. 5, pp. 408–422, 2019.

[27] C. J. Watkins *et al.*, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[28] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[29] "Openai gym cartpole-v1." https://gym.openai.com/envs/CartPole-v1/.

[30] "Openai gym lunarlander." https://gym.openai.com/envs/LunarLander-v2/.

[31] H. Watanabe *et al.*, "An fpga-based on-device reinforcement learning approach using online sequential learning," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 96–103, IEEE, 2021.