DARL: Distributed Reconfigurable Accelerator for Hyperdimensional Reinforcement Learning

Hanning Chen, Mariam Issa, Yang Ni, and Mohsen Imani
Department of Computer Science, University of California, Irvine, CA, USA
{hanningc, mariamai, yni3, m.imani}@uci.edu

ABSTRACT

Reinforcement Learning (RL) is a powerful technology to solve decision-making problems such as robotics control. Modern RL algorithms, i.e., Deep Q-Learning, are based on costly and resource hungry deep neural networks. This motivates us to deploy alternative models for powering RL agents on edge devices. Recently, brain-inspired Hyper-Dimensional Computing (HDC) has been introduced as a promising solution for lightweight and efficient machine learning, particularly for classification.

In this work, we develop a novel platform capable of real-time hyperdimensional reinforcement learning. Our heterogeneous CPU-FPGA platform, called DARL, maximizes FPGA's computing capabilities by applying hardware optimizations to hyperdimensional computing's critical operations, including hardware-friendly encoder IP, the hypervector chunk fragmentation, and the delayed model update. Aside from hardware innovation, we also extend the platform to basic single-agent RL to support multi-agents distributed learning. We evaluate the effectiveness of our approach on OpenAI Gym tasks. Our results show that the FPGA platform provides on average 20× speedup compared to current state-of-the-art hyperdimensional RL methods running on Intel Xeon 6226 CPU. In addition, DARL provides around 4.8× faster and 4.2× higher energy efficiency compared to the state-of-the-art RL accelerator while ensuring a better or comparable quality of learning.

ACM Reference Format:
Hanning Chen, Mariam Issa, Yang Ni, and Mohsen Imani. 2022. DARL: Distributed Reconfigurable Accelerator for Hyperdimensional Reinforcement Learning. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549437

1 INTRODUCTION

Reinforcement Learning(RL) is a sub-field in artificial intelligence that has amassed a lot of attention in the research community due to its ability to accomplish various tasks on new environments [1, 2]. In recent years, RL has proved to solve an impressive range of problems, including those that were previously unreachable for machines. Domains of tasks include those in system optimizations [3], Genomics [4], computer games [5, 6], and dynamic resource management [7]. Compared to its machine learning predecessors, both supervised and unsupervised machine learning methods, reinforcement learning does not need a massive compilation of data for training. Instead, an environment is defined to simulate the intelligent task that the model needs to learn, where the agent initially knows nothing of its environment. By interacting with its environment, the agent observes new states and their respective rewards and uses this acquired experience as feedback to learn how to improve and optimize its decision-making process. In Reinforcement Learning, there are two overarching approaches in doing this: the first, is policy-based learning [8], where the agent relies on a strategy to decide which state to subsequently advance to; the second, is value-based learning [9] where states are assigned a computed value to measure the expectation of attaining higher future rewards. The latter approach, includes Q-Learning, an algorithm that utilizes the Bellman Equation to compute the value for each state in the environment, named the Q-value [9]. At each time step, the Q-value for each possible next state is calculated to aid the agent in determining its next move. The agent iterates through this process until it reaches a terminating state or when the number of time steps in a session concludes.

As is known with Q-Learning algorithms, in large state and action spaces, the number of Q-values for each respective state-action pair is innumerable; thus, compiling the Q-values table requires a deep neural network to compute these values [10]. Neural networks are the current state-of-the-art for powering these algorithms despite the expensive computation needed to train these algorithms, especially during the backpropagation step [10]. Recently, Hyper-Dimensional Computing (HDC) has been introduced as a brain-inspired machine learning model for highly efficient and robust machine learning. The motivation behind HDC is based on how the human brain operates on high-dimensional data representations [11]. This observation leads to why HDC encodes objects with high-dimensional data representations, called *hypervectors* that store thousands of elements [12]. HDC imitates important functionalities of the human memory model with memory functions, such as storing and loading information. It also uses vector operations, which are computationally tractable and mathematically rigorous, in describing human cognition [13]. A couple advantageous elements of HDC include its ability to learn from single or a few shots of training, where data is quickly learned compared to iterative training algorithms. The second key advantage is how HDC is analogous to human-like learning and reasoning capabilities, which allows for for human-interpretable machine decisions.

Recent work uses this HDC model as an alternative to Deep Q-Learning for reinforcement learning tasks [14]. 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 [15, 16]. Despite the success, HDC-based RL still lack parallelism and require a large amount of resources on traditional cores [14]. Previous works already show that HDC related operations, such as hypervector multiplication, have a long execution time on the CPU [17, 18]. Furthermore, many domain-specific accelerators [19, 20, 21, 22, 23] have achieved great acceleration results of RL algorithms.

To the best of our knowledge, we propose the first FPGA-based hyperdimensional RL acceleration platform, called DARL (Distributed Accelerator for Hyperdimensional Reinforcement Learning). We developed this architecture and algorithm co-optimization to maximize an RL agent's learning throughput by realizing the best use of FPGA's resource utilization. Here are the main contributions of the paper:

- DARL is an online platform for accelerating hyperdimensional Q-Learning on FPGA. It first exploits hyperdimensional primitives to encode the agent's state into high-dimensional space. Then it approximates the dynamic programming equations by conducting a regression operation on the encoded state hypervector over the action model. Finally, the action with the highest Q-value will be

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10.
https://doi.org/10.1145/3508352.3549437

chosen and written into the replay buffer. The regression Q-model will also be updated. Therefore, our solution is a natural full-stack HDC accelerator by implementing both RL training and inference on the same platform.
- We conduct architecture optimization to accelerate on-chip hypervector operations. First, we design a hardware-friendly kernel encoding module to reduce on-chip resource utilization. Second, we fragmentized long hypervectors into small chunks and carefully designed a systolic array to accelerate vector to matrix multiplication. Third, we cut the original backpropagation style model update into two stages to reduce the accelerator's critical path and improve the learning throughput.
- We extend the original single-agent Q-Learning into multiple agents distributed Q-Learning. A lightweight network-on-chip IP is designed to coordinate different agents' learning progress. The on-chip high bandwidth memory is adopted to increase data-level parallelism. The RL agents' increment significantly enhances the learning convergence speed and model's robustness.

We evaluate the effectiveness of our approach on classic OpenAI Gym [24] 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 [14] that run on Intel Xeon 6226 CPU. On the Xilinx Alveo U280 accelerator card platform, drawing less than 20 Watt, DARL demonstrates 187972.9 IPS (inference per second) and 11053.7 IPS/W when deploying eight agents. Both the throughput and energy-efficiency of our design is around 4× better than the state-of-the-art RL accelerator [23]. DARL's throughput and energy efficiency reach 38.7k IPS and 5.2k IPS/W using only 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.

2 BACKGROUND AND MOTIVATION

2.1 Hyperdimensional Computing Overview

Stemming from theoretical neuroscience, HyperDimensional Computing (HDC) emerged as a short-term human memory model [11] that is conceptually motivated by the cerebellum cortex operating on high-dimensional representations of data, which originates from the extensive size of brain circuits. This thereby models the human memory with hypervectors, where the data is transformed into high-dimensional space [11]. Each of the dimensions in these hypervector models abstractly models a neuron's functionality in processing external stimuli. By choosing to work in high dimensional space, there exists a multitude of nearly orthogonal hypervectors. HDC exploits this property by using well-defined operations to combine hypervectors while maintaining their respective information with high probability [11]. In the training process, the encoded signal values are superimposed to produce a "model hypervector", which is the representation of the phenomenon of interest. The HDC defined association search operation is then used to perform the learning and cognitive computation over the encoded data. The similarity between the model hypervectors and query (i.e., new data point) is used to inform the model of the appropriate decision.

2.2 HDC-based Q-Learning

Q-Learning Overview The primary goal of Reinforcement Learning (RL) is training an agent's capabilities to find the reward-maximizing behavior when interacting with its environment [1]. 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 [9]. 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

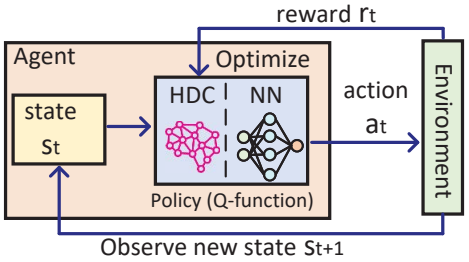

Figure 1: Q-Learning Overview

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.

Deep Q-Learning vs HDC-based Q-Learning One of the most critical components of every Q-Learning algorithm is the Q function. 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 [9]. 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 [25]. 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, which makes the deployment of DQN models on edge devices extremely difficult. Additionally, the loss gradient calculation and backpropagation limit the DQN model's performance improvement. To overcome these issues, there has been recent work [14] that 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, [14] shows that HDQL can achieve high learning rewards with limited memory access.

HDQL Acceleration on FPGA Although work in [14] shows HDQL's potential to solve RL tasks, it still has fundamental work to consider. There are two main issues that work [14] did not solve. The first problem is that work [14] only conducts experiments of HDQL on the CPU. However, as reported by previous HDC-related papers [26, 27, 17], HDC's high parallelism cannot be fully utilized on the CPU. The second problem is that work [14] only considers single-agent Q-Learning. Recent works [28, 29] already show that multiple agents' distributed training will significantly improve RL models' learning speed and robustness. In this paper, we first perform the Hardware-Software Co-design of a single agent HDQL (sHDQL) accelerator on CPU-FPGA heterogeneous platform. Then, in section 4, we extend the single-agent HDQL into multiple agents distributed HDQL(dHDQL) and design efficient accelerators on the same CPU-FPGA platform based on our algorithm. We name this RL agents' number varying platform as DARL (Distributed Accelerator for Hyperdimensional Reinforcement Learning).

3 DARL ACCELERATION PLATFORM

3.1 DARL Overview

Figure 2 shows the DARL platform's top architecture. 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,

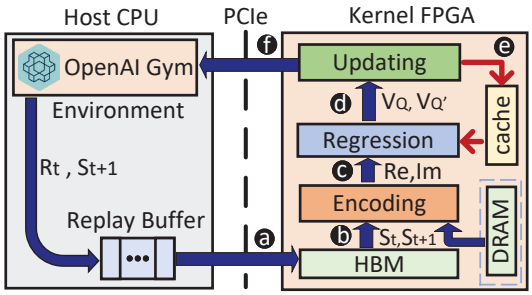

Figure 2: HDQL acceleration overview on CPU-FPGA Platform.

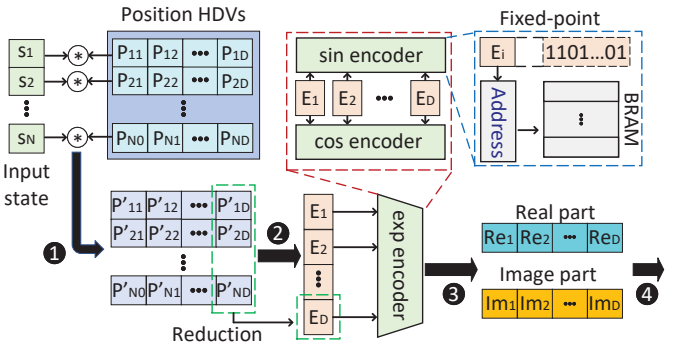

Figure 3: Encoding layer architecture design.

the host CPU will offload corresponding state, action, and reward data to the FPGA kernel via PCIe communicationas as shown in Figure 2 ⓐ. 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 ($\vec{s_t}$) 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($f(x) = e^{jx}$), 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 (ⓔ). More details of these three layers will be discussed from section 3.2 to section 3.4.

3.2 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 3 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 hypervecters will be stored on-chip as a position hypervector matrix (**P**). **P**'s dimension will be $N \times D$. In process ❶, each row vector of matrix **P** will be multiplied with its corresponding state vector's element:

$$\vec{P_i'} = s_i * \vec{P_i} \quad i \in [1, N] \tag{1}$$

In process ❷, the reduction operation will be applied over matrix **P'** row direction to generate hypervector $\vec{E}$:

$$E_j = \sum_{i=0}^{N} P'_{i,j} \quad i \in [1, N] \; and \; j \in [1, D] \tag{2}$$

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 $\vec{E}$:

$$E_i' = e^{jE_i} \quad i \in [1, D] \tag{3}$$

To avoid computing over these complex numbers on the FPGA, our design divides the complex hypervector $\vec{E'}$ into its real part ($\vec{Re}$) and imaginary part($\vec{Im}$), as is shown below:

$$E_i' = Re_i + jIm_i \quad i \in [1, D] \tag{4}$$

$$Re_i = cos(E_i) \quad i \in [1, D] \tag{5}$$

$$Im_i = sin(E_i) \quad i \in [1, D] \tag{6}$$

Furthermore, after passing the hypervector into the exp encoder IP, two hypervectors: $\vec{Re}$ and $\vec{Im}$ will be generated (❸) as the encoded result. Now the input state vector is mapped into hyperspace. The last process ❹ in the encoding layer is to pass the real ($\vec{Re}$) and imaginary partitions of the hypervectors ($\vec{Im}$) into the regression layer.

In this section, we elaborate on the implementation of our exponential encoder IP on FPGA. Inside each exp encoder IP, there are Sine and one Cosine encoder IPs. For the FPGA hardware design, there are many efficient methods to implement triangle functions, such as Taylor Expansion or using Vitis HLS math function. To save on-chip resources utilization and reduce computing latency, we choose to use what we call the **triangle codebook** method. As we mentioned in section 3.1, for each element of the hypervector, its precision is a 32-bits fixed point number, which means we can pre-compute all its possible sin and cos values on the local CPU and load it into the FPGA on-chip storage such as BRAM or URAM. Here, 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 signed 32-bits 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 includes reducing resources utilization, especially LUT and DSP, and saving calculation time.

3.3 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 both: $A \times D$. 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 4 processes ⓐ and ⓑ. The purpose of doing this is to reduce data transmission time and realize online learning. At the

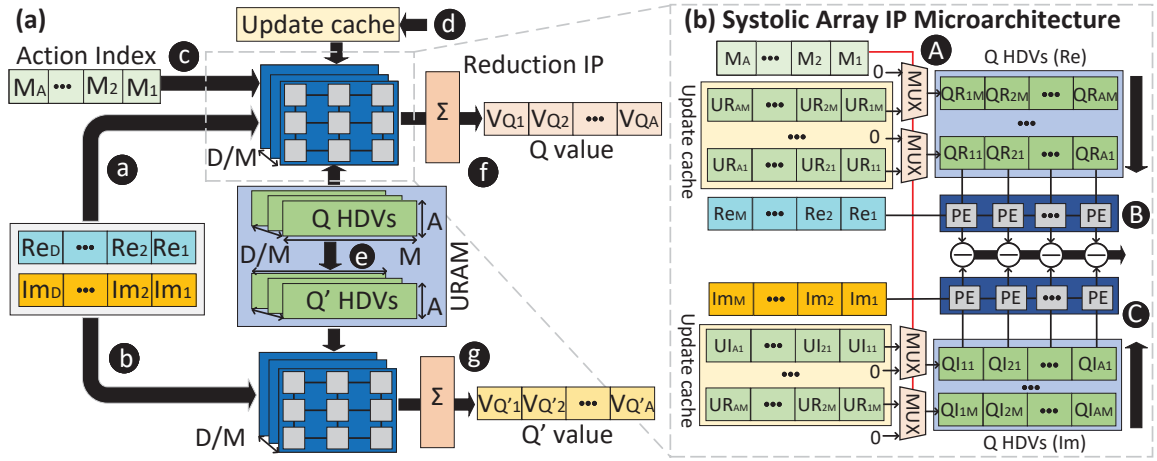

Figure 4: (a) Regression layer architecture design. (b) Systolic Array IP microarchitecture

starting point of each episode, the model **Q** will flush **Q'**, as is shown in process ❹. 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. To reduce the FPGA accelerator's critical path, the model update matrix is stored inside an update cache. For each time step, the host CPU will load the action index vector: $\vec{M}$ into the kernel FPGA. 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 ❸ and ❹. The model Q's updating equation is shown as below:

$$QR_{i,j} = \begin{cases} QR_{i,j} + UR_{i,j} & M[i] == True \\ QR_{i,j} & M[i] == False \end{cases} \quad i \in [1, A] \text{ and } j \in [1, D] \tag{7}$$

$$QI_{i,j} = \begin{cases} QI_{i,j} + UI_{i,j} & M[i] == True \\ QI_{i,j} & M[i] == False \end{cases} \quad i \in [1, A] \text{ and } j \in [1, D] \tag{8}$$

Here **QR** represents the real part of the Q matrix and **QI** represents the imaginary part of the Q matrix. **UI** and **UR** represent real and imaginary parts of the update matrix. Analogous to the memory hierarchy design, people use a writing buffer to delay the synchronization process between the main memory and cache hierarchy. Using the update cache will allow for the action hypervector update to look similar to the forward path, making the Vitis_HLS synthesis and placement process significantly easier to schedule. The structure of the update cache is like a hardware FIFO. The input of the FIFO comes from the updating layer, as is shown in Figure 4 process ❷ and the output of the FIFO is in the regression layer. More details of the update matrix generation will be covered in section 3.4.

In Figure 4, 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 4, 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** × **M** matrix. This process will happen for both real and imaginary parts, as is shown for process ❽ and ❾. 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. The calculation formulas of Q value vector $\vec{V_Q}$ and Q' value vector $\vec{V_{Q'}}$ are shown below:

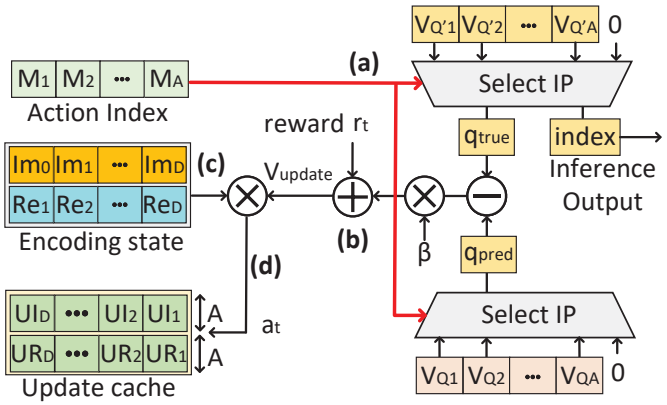

Figure 5: Updating layer architecture design. (a)Q and Q' value selection. (b)Update value Calculation. (c)Multiply the update value with encoded state hypervector. (d) Store update hypervector into Update cache

$$\vec{V_Q} = \sum_{k=1}^{\frac{D}{M}} \frac{\vec{Re_k} \cdot QR_k{}^T}{||Re_k|| * ||QR_k||} - \frac{\vec{Im_k} \cdot QI_k{}^T}{||Im_k|| * ||QI_k||} \quad k \in [1, \frac{D}{M}] \tag{9}$$

$$\vec{V_{Q'}} = \sum_{k=1}^{\frac{D}{M}} \frac{\vec{Re_k} \cdot QR'_k{}^T}{||Re_k|| * ||QR'_k||} - \frac{\vec{Im_k} \cdot QI'_k{}^T}{||Im_k|| * ||QI'_k||} \quad k \in [1, \frac{D}{M}] \tag{10}$$

Here QR' and QI' are the real and imaginary parts of model **Q'**. As the synthesis and placement result show, cutting hypervectors into partitions significantly saves on-chip resource utilization. Finally, the generated $\vec{V_Q}$ and $\vec{V_{Q'}}$ will be passed to the updating layer, as is shown in process ❻ and ❼.

3.4 Updating Layer Architecture

In Figure 5.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 $\vec{M}$. The mathematical function of select IP is shown as below:

$$index = \begin{cases} argmax_i V_Q[i] & if\ \forall_{i\in[1,A]}\ M[i] == False \\ -1 & otherwise \end{cases} \quad (11)$$

$$q_{pred} = \begin{cases} V_Q[i] & if\ \exists_{i\in[1,A]}\ M[i] == True \\ 0 & otherwise \end{cases} \quad (12)$$

$$q_{true} = \begin{cases} V_{Q'}[i] & if\ \exists_{i\in[1,A]}\ M[i] == True \\ 0 & otherwise \end{cases} \quad (13)$$

In Figure 5.b, the update value is calculated as shown below:

$$V_{update} = (q_{true} - q_{pred}) * \beta + r_t \quad (14)$$

Here β is the learning rate and r_t is the environment reward. The r_t is also sampled from the replay buffer.

In Figure 5.c and Figure 5.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 $\vec{M}$, as shown below:

$$\vec{UR_i} = \begin{cases} \vec{Re} * V_{update} & M[i] == True \\ \vec{UR_i} & M[i] == False \end{cases} \quad i \in [1, A] \quad (15)$$

$$\vec{UI_i} = \begin{cases} \vec{Im} * V_{update} & M[i] == True \\ \vec{UI_i} & M[i] == False \end{cases} \quad i \in [1, A] \quad (16)$$

Specifically, in Figure 5.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 as discussed in section 3.3. For example, suppose during the training process that at time step $t + 2$, the action index a_{t+2} is equal to a_t, then the corresponding action hypervector of model Q will be updated as is shown below:

$$\vec{QR_{a_{t+2}}} = \vec{QR_{a_{t+2}}} + \vec{UR_{a_{t+2}}} \quad where\ M[a_{t+2}] = True \quad (17)$$

$$\vec{QI_{a_{t+2}}} = \vec{QI_{a_{t+2}}} + \vec{UI_{a_{t+2}}} \quad where\ M[a_{t+2}] = True \quad (18)$$

By cutting the backpropagation update process into two stages, as is shown by Vitis HLS synthesis result, the critical path of the kernel accelerator is shortened. More details of the performance improvement will be presented in section 6.4.

4 DISTRIBUTED HDC-BASED Q-LEARNING

In section 3, we present the details of the architecture design of the single-agent version of DARL platform (DARL$_1$). This section will introduce how to realize multiple agents distributed learning on a single FPGA chip. We utilize high bandwidth memory (HBM) to achieve multiple agents' data-level parallelism. A lightweight network-on-chip (NoC) IP is also designed to accelerate the synchronization process between multiple agents.

4.1 Top Distributed On-chip Learning Architecture

The top architecture of the multiple agents DARL platform is shown in Figure 6. Suppose there are a total of N_a RL agents, and each agent has its own HBM channel. Each Xilinx HBM channel has two pseudo channels (PS). Here we index them as PS0 and PS1. As is shown in Figure 6.b, the host CPU will copy the training batch or inference state into the HBM PS0. Each agent will then read these tuples from PC0, as

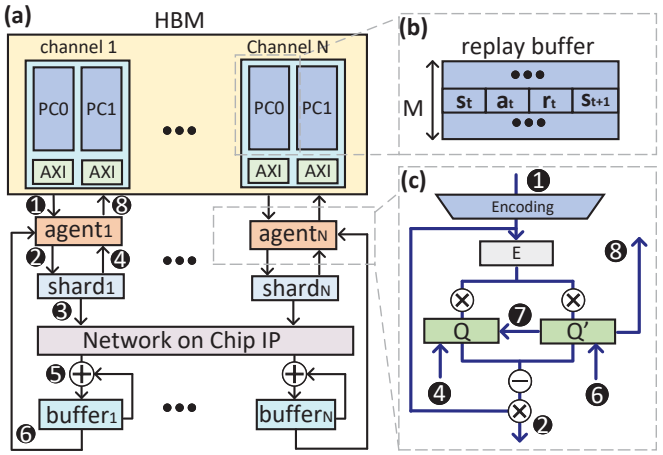

Figure 6: Distributed HDC-based Q-Learning Platform

is shown in Figure 6 ❶. During the inference stage, each agent will pass the action index of current time step (a_t) back to the host CPU as shown in ❽.

Inside each agent of Figure 4, there still exists encoding, regression, and update layers. For better illustration, we also present the simplified version of each agent's internal architecture in Figure 4.c. As shown in process ❷, each agent has its own update cache: **shard**, which is used to store the agent's current action hypervector update. The updating process of each agent's model Q is shown as process ❹ in Figure 4. The main difference between distributed Q-Learning and DARL$_1$ is the synchronization of the different agents' training update. We assume that during the training stage, each agent works independently, which means each agent's model Q will be updated independently. However, during the inference stage, each agent's models, Q and Q', need to be synchronized. The bundled mode concept is therefore introduced by the previous distributed learning work [29]. Here, the bundled mode indicates that each agent has its own local model Q and a common Q′ target model. The common Q′ means that each agent will keep a replica of model Q′ so that the inference stage can be independent from each other. Despite each agent not being updated Q′ directly during the training process, for later target Q′ update synchronization, a simplified network-on-chip (NoC) IP is added to conduct the average and scatter operation for each agent's update hypervector as is shown in the process ❸ and ❺. This average and scatter operation is called an **AllReduce** operation. More discussion of this AllReduce operation will be included in section 4.2.

After the last time step of the current training stage, the synchronized action hypervector update will be added to each agents' target model: Q′ as is shown in process ❻. Also, the model Q' will flush the agent's local model Q as is shown in process ❼. In this case, at the beginning of the next training-inference cycle, all agents' local models: Q and target model: Q' will be synchronized. As we will see in section 6, the learning speed and target model's robustness will be significantly improved due to multiple agents' participation.

4.2 AllReduce via Network On Chip IP

AllReduce operator is widely adopted in distributed training platforms to synchronize different model parameters [30, 31]. Here we define the AllReduce operator as averaging all agents' action hypervector update stored in the shard and scattering it over all agents' local buffer **U**. After the AllReduce operator, each agent's buffer will have a copy of the action hypervector update. After the last time step of the training stage, this update will be added to each agent's target **Q'** model. The

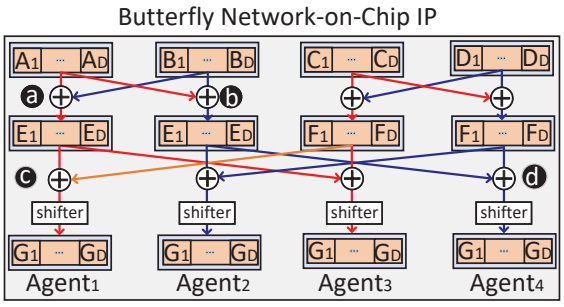

Figure 7: Four Nodes AllReduce Butterfly Notwork.

equation for the AllReduce operator is shown below:

$$U_i[j] = \frac{1}{N_a} \cdot \sum_{k=0}^{N_a} shard_k[j] \quad i \in [1, N_a] \; and \; j \in [1, A] \tag{19}$$

Here i is the agent index and j is the action index. Both shard and local buffer are on-chip storage and will be flushed to 0 at the beginning time step of the training stage.

On-chip AllReduce operator is normally realized by designing a specific network-on-chip(NoC) IP. For general on-chip NoC design, people typically focus on network topology, router microarchitecture, and flow control mechanisms. In this paper, since we mainly focus on RL algorithm acceleration and only want to realize multiple agents' target model synchronization, a straightforward NoC IP is designed, which is shown in Figure 6. The NoC topology that we selected for our design is the butterfly topology [32]. Compared to other popular topologies such as ring, mesh, or torus, the butterfly is easier to implement on a resources-constrained FPGA platform [33, 21]. The process of AllReduce operator inside the NoC IP is illustrated in Figure 7. Assume each agent has a unique vector denoted $\vec{A}$, $\vec{B}$, $\vec{C}$, $\vec{D}$, where each vector's dimension is D. In this case, each agent will first add its own vector with its neighbor's vector and pass it to the next layer. So in Figure 7 ⓐ and ⓑ we have:

$$E_i = A_i + B_i \quad i \in [1, D] \tag{20}$$

$$F_i = C_i + D_i \quad i \in [1, D] \tag{21}$$

Until now, each agent has exchanged its model with its neighbor. But since we have 4 nodes, the exchange process will continue. In process ⓒ, $Agent_1$ will exchange its vector with $Agent_3$, and in process ⓓ, meanwhile $Agent_4$ will exchange its vector with $Agent_2$. So we have:

$$G_i = \frac{E_i + F_i}{4} = \frac{A_i + B_i + C_i + D_i}{4} \quad i \in [1, D] \tag{22}$$

It is now evident to observe that each agent's vector is shared, thus, the AllReduce operation has been completed. Here we only present the process of 4 agent nodes AllReduce operator, however, the same structure could be pursued for any number of agents.

5 RL AGENT'S THROUGHPUT CALCULATION

This section introduces the RL model learning throughput's definition and calculation. Unlike regular supervised or unsupervised learning, the throughput of RL should consider both the training and inference stage. We define the throughput as the number of inferences processed per second (IPS), which is also widely adopted by previous work [20, 21, 23]. Suppose there are a total of N_{agent} agents in the DARL platform, and the total learning episodes length is E. Each agent's environment interaction times in the i^{th} episode is T_i. In RL terminology, we also call T_i the trajectory length. Based on the ϵ greedy algorithm introduced in section 3.1, we also defined the kernel FPGA's i^{th} episode iteration cycles for inference and training as $T_i^{'}$ and $T_i^{''}$ respectively. Here we also define the ratio between $T_i^{'}$ and $T_i^{''}$ as the training frequency(β). The batch size for each training step is M. Suppose that the data communication times between the host CPU and kernel FPGA for inference and training are T_{comm1}(s) and T_{comm2}(s) respectively. Each step's kernel runtime is F(s). We also use T_{env}(s) to represent the agents' environment interaction time. T_{other} represents other execution latency on CPU such as Xilinx Runtime Platform(XRT) initiation time. Equation 23 24 25 show the throughput(IPS) formal mathematical calculation process.

Table 1: Resource Utilization and Performance on Alveo U280

	CartPole		LunarLander	
Config[1]	$DARL_1$	$DARL_8$	$DARL_1$	$DARL_8$
LUT	73.1K (6%)	866.7K (73%)	117416 (8%)	988.2K (84%)
BRAM	276 (6%)	1425(30%)	546 (12%)	2825 (65%)
URAM	79 (8%)	431(48%)	143 (14%)	431 (48%)
FF	38047 (1%)	329.1K(11%)	42508 (1%)	367.7K(12%)
DSP	17	17	17	17
f (MHz)[2]	171 MHZ	171 MHz	171 MHz	171 MHz
L (cycle)[3]	417	547	421	624

[1] Configuration of DARL [2] The frequency [3] The latency

$$IPS = \frac{\sum_{i=0}^{E} N_{agent} \times T_i}{T_{inf} + T_{train} + T_{other}} \quad where \tag{23}$$

$$T_{inf} = T_{comm1} + \sum_{i=0}^{E} (T_i^{'} \times F + T_{env} \times T_i) \tag{24}$$

$$T_{train} = T_{comm2} + \sum_{i=0}^{E} T_i^{''} \times M \times F \tag{25}$$

6 EVALUATION

6.1 Experimental Setup

We develop full DARL library using two modules: (1) an optimized software implementation using Python library supported encoding and learning phase of our reinforcement learning, (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 [24], including CartPole [5] and LunarLander [6]. We evaluated our design's performance in three aspects. The first is our HDC-based RL algorithm's accuracy, the second is total execution latency, and the third is the learning throughput and energy efficiency. In the whole section 6, we use $DARL_1$ and $DARL_8$ to represent the DARL platform with single and eight agents.

6.2 Single vs. Multi-Agent: Resource Utilization

Table 1 reports the accelerator's resource utilization of HDQL running on the Xilinx Alveo U280 platform. The synthesis and implementation tool that we use is Xilinx Vitis HLS and Vivado 2021.2. Here, the hypervector dimension is 2048 (2K), and each tuple's precision is a fixed-point 32-bit. As is shown by previous work [34], the dimension of the hypervector will have a significant influence on the model's accuracy, which means that in order to achieve high rewards for learning tasks, the dimension of the hypervector cannot be very low. 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 1, the resource utilization for those two tasks is different.

Table 1 provides detailed resource utilization for our designed accelerator with varying number of agents. We found that Xilinx Alveo U280 can support at most eight agents without sacrificing performance. However, we believe a board with more resources, such as

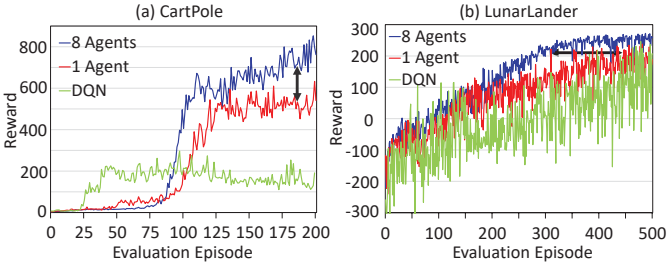

Figure 8: DARL platform agents' rewards over episodes.

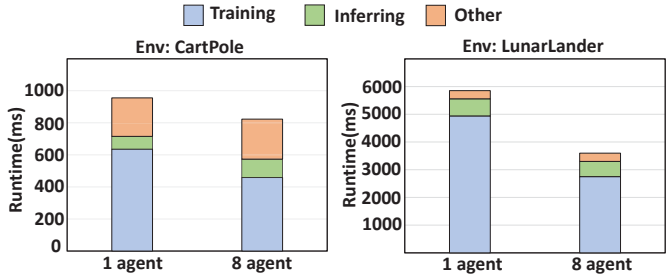

Figure 9: Runtime breakdown of DARL platform for CartPole and LunarLander tasks.

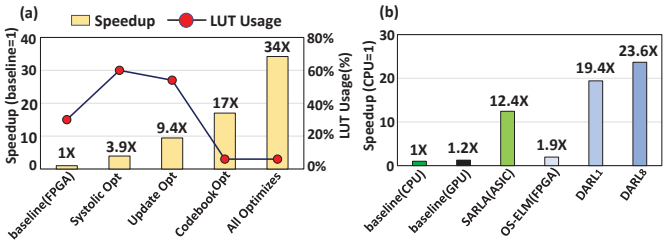

Figure 10: DARL's performance for OpenAI Gym CartPole task. (a) Impact of different optimization techniques on DARL resource utilization and speedup. (b) Different platform's runtime comparison. Here we assume both SARLA [22] and OS-ELM [35]'s rewards accumulation are the same as ours.

Xilinx VCU128 FPGA could conduct larger-scale distributed learning. 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. To increase learning throughput and reduce the number of learning episodes, a distributed training model with a decent number of agents, based on their FPGA board's resource condition, can be selected. In Table 1, it is also apparent that, after using eight agents for distributed learning, the FPGA on-chip resources are almost fully utilized, with the exception of DSP. Compared to traditional deep learning, one of the most critical strengths of HDC is that the model update is not based on gradient descent. 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 the energy efficiency improvement, which will be discussed in section 6.5. In Figure 10.a, we also present the three optimization's influence over $DARL_1$'s resource utilization and speedup. Reducing the accelerator's critical path and designing a lightweight kernel encoder allows for DARL to achieve high-performance learning throughput speed while requiring affordable resource utilization.

The last part that we want to discuss in this section is the single iteration latency on FPGA. In Table 1, we only cover the on-chip latency which means the latency is only related to processes ⓑ, ⓒ, ⓓ, ⓔ in Figure 2. Other important RL parts, such as the interaction with the environment or the passing of data from CPU to FPGA via AXI DMA, are not included in Table 1. We will provide the total execution latency for both tasks in section 6.4. Compared to single-agent learning, distributed learning takes more clock cycles to finish one step iteration. On-chip muti-agents synchronization makes the single-step latency increase. However, compared to previous work [20] synchronizing agents' learning reward on CPU, the latency overhead of conducting the AllReduce operation in our design is small due to the butterfly NoC IP's support. In section 6.3, we will show that distributed training will significantly improve the RL agents learning throughput.

6.3 Performance: Algorithm Accuracy

Figure 8 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 that we selected for CartPole is 8 and for LunarLander, 16. We also test DQN [25] 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.

Compared to single-agent learning, multiple agents distributed learning achieves significantly higher rewards within the same episode for both tasks, as shown by the black arrow in Figure 8.a. To achieve OpenAI Gym suggested accumulated rewards target, such as 200 for the LunarLander task, multi-agents need fewer episodes than single-agent training, as shown by the black arrow in Figure 8.b. In addition to convergence speed improvements, we also observe that for complex control task such as LunarLander, distributed training significantly improves the learning reward's steadiness. As is shown in Figure 8, the fluctuation of rewards for multiple agents is much lower than for single agents. We believe multiple agents makes the learning model's robustness much higher.

6.4 Performance: Execution Time

Figure 9 presents our design's execution time for accumulating target rewards for two OpenAI Gym environments. Specifically, for the CartPole environment, the target reward is 500, and for the LunerLander environment is 200. In Figure 9, we can see that the distributed training significantly speeds up the model's learning speed. In Figure 9, we also provide the execution time breakdown. Here we cut the total execution time into three parts: training execution time(Training), inferring execution time(Inferring), and Vitis platform's overhead(Other). The Inferring part here also includes agents' interaction with the environment and receiving a reward from the environment. The Other part includes all latency caused by the Vitis Xilinx runtime(XRT), such as OpenCL initialization time, buffer allocation time, and CPU-FPGA PCIe communication time. For small tasks, such as Cartpole, the XRT platform overhead portion is considerable since the total learning episode is small. However, its portion is much smaller for a more complicated task, such as LunarLander.

We also implement the HDC-based Q-Learning on Intel Xeon 6226 for comparison. The CPU's execution times for the CartPole environment and LunarLander environment are 19.41s and 1568.3s respectively. Our single-agent accelerator achieves around 19x and 260x speedup compared to CPU for CartPole and LunarLander, respectively. With the more complicated tasks, such as action space and state dimension increasing, DARL's speedup becomes more apparent. We also compare DARL with other platforms' acceleration result in Figure 10.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.

Table 2: Comparison Table with Previous RL Acceleration Works

	ASPLOS'19 [20]	FCCM'20 [21]	ICCAD'20 [22]	IPDPSW'21 [35]	DAC'21 [23]	$DRAL_1$	$DRAL_8$
Platform	Xilinx VCU1525	Alveo U200	ASIC	PYNQ-Z1	Alveo U50	Alveo U280	
Clock	180MHz	285MHz	800MHz	100MHz[1]	164MHz	171MHz	
Algorithm	A3C	PPO	A3C[2]	DQN	DDPG	sHDQL	dHDQL
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	512 KB
Throughput	12849.1 IPS	6823.2 IPS	+	+	38779.8 IPS	36597.1 IPS	187972.9 IPS
Energy Efficiency [3]	141.7 IPS/W	-	+	+	2638.0 IPS/W	5256.3 IPS/W	11053.7 IPS/W

[1] 100MHz is Zynq FPGA PL part's frequency. For PS ARM core part, the frequency is 650MHz. [2] Work in [22] supports not just A3C. [3] Only accelerator's energy efficiency.
[−] Paper didn't provide relate information. [+] Paper provide related information and we discussed it in section 6.4 and section 6.5.

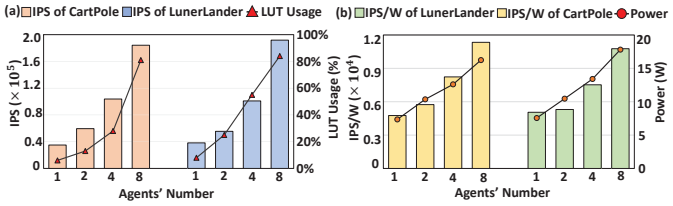

Figure 11: (a) Learning throughput(bar) and LUT usage(line) with different number of agents. (b) Energy efficiency(bar) and power consumption(line) with different number of agents.

6.5 Training Throughput and Energy Efficiency

In Figure 11.a, we present the DARL platform's learning throughput(IPS) and lookup table(LUT) usage, which varies by the number of agents. As mentioned in section 5, the throughput represents the ratio of the total number of collected samples to the entire system runtime. In Figure 11.a, when increasing the number of agents, the throughput also increases. The maximum throughput we can achieve when using eight agents is 184158.5 and 191787.4, respectively, for CartPole and LunarLander environments. In Figure 11.a, we also notice that the platform's throughput is nonlinear to the number of agents(N_{agent}). As mentioned in sections 6.3 and section 6.4, distributed RL decreases the total learning episodes, which means the trajectory length (T) also decreases. The equation 23 in section 5 shows that increasing N_{agent} and decreasing T causes the throughput's increment to be nonlinear to N_{agent}'s increment.

Figure 11.b reported our platform's energy efficiency and power consumption change varying different number of agents. We use Xilinx Power Estimator(XPE) to estimate the kernel accelerator's power and ignore the host CPU's power consumption. We did this for the later part's fair comparison with previous RL acceleration works. Due to using HBM, our accelerator's power consumption is relatively higher than previous ASIC work [22] but still less than previous FPGA acceleration work [23] since HDC's nature strength reduces the kernel's DSP usage.

Table 2 summarizes the comparison of our design with previous RL acceleration work published at the top conference. Here $DARL_1$ and $DARL_8$ represent the DARL platform with single and eight agents. The throughput and energy efficiency of $DARL_1$ and $DARL_8$ included in the Table 2 are the average of CartPole and LunarLander's results in Figure 11. Our design shows around 4x improvement over the state-of-the-art RL FPGA accelerator [23] for throughput and energy efficiency.

7 RELATED WORKS

Reinforcement Learning Accelerator Accelerating reinforcement learning (RL) algorithms on Domain Specific Architectures(DSA), such as FPGA and ASIC, have recently amassed considerable attention [36]. Generally speaking, RL algorithms can be divided into two categories, tabular RL and deep RL. Traditional acceleration work focuses on tabular RL algorithms acceleration [37, 38, 39]. However, the key component of tabular RL, the Q table, is prone to becoming excessively large which prevents tabular RL from handling complicated tasks. To overcome this challenge, a number of recent works have been proposed to accelerating neural network based RL(Deep RL) [40, 35]. Recent years, many famous RL algorithms such as DQN [25], DDPG [41], PPO [8] have been accelerated on the FPGA platform [40, 35, 21, 23]. However, all these works focused on single-agent RL acceleration. Multi-agents distributed RL has been proved to be successful by previous works [29, 42, 28]. With the improvements to single chip computing capability, on-chip distributed RL acceleration has been proposed recently [20, 22]. However, work [20] didn't conduct multiple agents on-chip synchronization, restricting the RL model's learning speed. Work [22] instead only focuses on the interconnection part of the different agents but focuses little on the single agent's acceleration architecture.

Hyperdimensional Computing Hyperdimensional computing (HDC) was first introduced by neuroscientist P. Kanerva [11]. One of the important strengths of HDC is that it does not have deviation based backpropagation. This makes HDC acceleration on hardware platform, such as FPGA, popular [17]. Prior works have applied HDC into diverse cognitive tasks, such as robotics [43], genome pattern matching [44, 45], and speech recognition [46]. Recently people have successfully applied HDC to solve RL tasks [14]. The core idea is to use HDC based regression functions [47] to approximate the Q function of the RL agent. Work [14] shows HDC's potential in the RL area, but is only preliminary work. To maximize HDC's full strength, the hardware acceleration of HDC-based RL algorithms(HDRL) is necessary. However, no prior work has attempted to accelerate HDRL on FPGA or other hardware platforms. Hence, we propose the DARL platform in this paper as an efficient FPGA-based hardware acceleration of HDC-based RL algorithms.

8 CONCLUSION

In this paper, we develop a novel platform capable of real-time hyperdimensional reinforcement learning. Our heterogeneous CPU-FPGA platform, called DARL, maximizes FPGA's computing capabilities by applying several hardware optimizations to hyperdimensional computing, including hardware-friendly encoder IP, the hypervector chunk fragmentation, and the delayed model update. Aside from hardware innovation, we also extend the platform from basic single-agent RL to support multi-agents distributed learning. We evaluate the effectiveness of our approach on OpenAI Gym tasks.

ACKNOWLEDGEMENTS

This study is supported by the National Science Foundation (NSF) #2127780, Semiconductor Research Corporation (SRC) Task No. 2988.001, Department of the Navy, Office of Naval Research, grant #N00014-21-1-2225 and #N00014-22-1-2067, Air Force Office of Scientific Research, grant #22RT0060, and generous gifts from Cisco and Xilinx.

REFERENCES

[1] 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.

[2] P. Henderson *et al.*, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, 2018.

[3] N. Jay *et al.*, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*, PMLR, 2019.

[4] M. Imani *et al.*, "Control of gene regulatory networks using bayesian inverse reinforcement learning," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 16, no. 4, pp. 1250–1261, 2018.

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

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

[7] Y. He *et al.*, "Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach," *IEEE Communications Magazine*, vol. 55, no. 12, pp. 31–37, 2017.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

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

[10] V. François-Lavet *et al.*, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.

[11] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive computation*, vol. 1, no. 2, pp. 139–159, 2009.

[12] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circuits and Systems Magazine*, vol. 20, no. 2, pp. 30–47, 2020.

[13] P. Poduval, A. Zakeri, F. Imani, H. Alimohamadi, and M. Imani, "Graphd: Graph-based hyperdimensional memorization for brain-like cognitive learning," *Frontiers in Neuroscience*, p. 5, 2022.

[14] Y. Ni, D. Abraham, M. Issa, Y. Kim, P. Mercati, and M. Imani, "Qhd: A brain-inspired hyperdimensional reinforcement learning algorithm," *arXiv preprint arXiv:2205.06978*, 2022.

[15] A. Hernandez-Cane, N. Matsumoto, E. Ping, and M. Imani, "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.

[16] Z. Zou, Y. Kim, F. Imani, H. Alimohamadi, R. Cammarota, and M. Imani, "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.

[17] M. Imani, Z. Zou, S. Bosch, S. A. Rao, S. Salamat, V. Kumar, Y. Kim, and T. Rosing, "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.

[18] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.

[19] A. Samajdar, P. Mannan, K. Garg, and T. Krishna, "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.

[20] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "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.

[21] Y. Meng, S. Kuppannagari, and V. Prasanna, "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.

[22] Y. Wang, M. Wang, B. Li, H. Li, and X. Li, "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.

[23] J. Yang, S. Hong, and J.-Y. Kim, "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.

[24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[25] 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.

[26] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, "Dual: Acceleration of clustering algorithms using digital-based processing in-memory," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 356–371, IEEE, 2020.

[27] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 53–62, 2019.

[28] H. Y. Ong, K. Chavez, and A. Hong, "Distributed deep q-learning," *arXiv preprint arXiv:1508.04186*, 2015.

[29] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, *et al.*, "Massively parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1507.04296*, 2015.

[30] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 81–92, IEEE, 2020.

[31] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, vol. 2, 2017.

[32] P. Lotfi-Kamran, B. Grot, and B. Falsafi, "Noc-out: Microarchitecting a scale-out processor," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 177–187, IEEE, 2012.

[33] G. S. Malik and N. Kapre, "Enhancing butterfly fat tree nocs for fpgas with lightweight flow control," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 154–162, IEEE, 2019.

[34] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, 2016.

[35] H. Watanabe, M. Tsukada, and H. Matsutani, "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.

[36] M. Rothmann and M. Porrmann, "A survey of domain-specific architectures for reinforcement learning," *IEEE Access*, vol. 10, pp. 13753–13767, 2022.

[37] L. M. Da Silva, M. F. Torquato, and M. A. Fernandes, "Parallel implementation of reinforcement learning q-learning technique for fpga," *IEEE Access*, vol. 7, pp. 2782–2798, 2018.

[38] S. Spano, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, A. Nannarelli, and M. Re, "An efficient hardware implementation of reinforcement learning: The q-learning algorithm," *Ieee Access*, vol. 7, pp. 186340–186351, 2019.

[39] R. Rajat, Y. Meng, S. Kuppannagari, A. Srivastava, V. Prasanna, and R. Kannan, "Qtaccel: A generic fpga based design for q-table based reinforcement learning accelerators," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 323–323, 2020.

[40] J. Su, J. Liu, D. B. Thomas, and P. Y. Cheung, "Neural network based reinforcement learning acceleration on fpga platforms," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 68–73, 2017.

[41] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[42] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International Conference on Machine Learning*, pp. 1407–1416, PMLR, 2018.

[43] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, "Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception," *Science Robotics*, 2019.

[44] 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 Design Automation Conference (DAC)*, pp. 781–786, IEEE, 2021.

[45] Z. Zou, H. Chen, P. Poduval, Y. Kim, M. Imani, E. Sadredini, R. Cammarota, and M. Imani, "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.

[46] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *2017 IEEE international conference on rebooting computing (ICRC)*, pp. 1–8, IEEE, 2017.

[47] 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.