Associative Memory Based Experience Replay for Deep Reinforcement Learning

Mengyuan Li[1]* Arman Kazemi[1] Ann Franchesca Laguna[2] X. Sharon Hu[1]*
[1]Department of Computer Science and Engineering, University of Notre Dame, USA
[2] De La Salle University, Manila, Philippines
*mli22, shu@nd.edu

ABSTRACT

Experience replay is an essential component in deep reinforcement learning (DRL), which stores the experiences and generates experiences for the agent to learn in real time. Recently, prioritized experience replay (PER) has been proven to be powerful and widely deployed in DRL agents. However, implementing PER on traditional CPU or GPU architectures incurs significant latency overhead due to its frequent and irregular memory accesses. This paper proposes a hardware-software co-design approach to design an associative memory (AM) based PER, AMPER, with an AM-friendly priority sampling operation. AMPER replaces the widely-used time-costly tree-traversal-based priority sampling in PER while preserving the learning performance. Further, we design an in-memory computing hardware architecture based on AM to support AMPER by leveraging parallel in-memory search operations. AMPER shows comparable learning performance while achieving 55× to 270× latency improvement when running on the proposed hardware compared to the state-of-the-art PER running on GPU.

1 INTRODUCTION

Deep reinforcement learning (DRL) combining reinforcement learning and deep learning is a powerful framework for agents to learn to make decisions based on trial and error. DRL can be used in many applications such as gaming, robotics and other automated systems [6]. Some DRL methods learn offline, while others conduct learning online where an agent learns as it interacts with the environment. Online DRL is preferred when the environment is complex and changes often. It is highly desirable for Online DRL to satisfy certain real-time latency constraints. Deep Q-network (DQN), first introduced by Google DeepMind in [16], is a popular, model-free, online, off-policy DRL method.

In DQN, an agent learns through past experiences which are described by state transitions, rewards, and actions. A DQN agent is comprised of three main components: (1) an *action network* which determines the action at each time step for a given input state, (2) a *target network* which learns from past experiences, and (3) an *experience replay (ER) memory* which stores experiences and generates specific experiences for the target network as training input. The structure of target and action networks can be multilayer perceptrons (MLPs) or convolution neural networks (CNNs) [16].

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

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549387

For complex environments with many states, the ER memory can be very large, and it can take a significant amount of time to update the memory and generate new experiences. Through detailed profiling of several open-source DQNs running on GPU, we find that ER operation (sampling experiences) can take more than 55% of the total DQN execution time. Though there is an abundance of work on accelerating neural networks used in the action/target network, few works have considered accelerating the ER related operations. In order to meet real-time latency requirements for online deployment of DQNs, it is critical to devise techniques to accelerate the ER operations in DQN, which is our focus.

The key operation supported by ER memory is sampling a small subset of the stored experiences as the training data for the target network at each time step. ER memory can be very large (e.g. on the order of 10^6 entries) since the experiences at many past time steps may need to be stored. Hence sampling experiences faces the memory-wall [7] challenge for CPU and GPU implementations. Also, sampling techniques involve non-trivial calculations and can significantly impact the learning performance and speed. Uniform sampling was used in the earlier DQNs but its performance was not high. Prioritized experience replay (PER) [19], deploying priority sampling technique, is widely used in the state-of-the-art DQN implementations like Rainbow [8] and Agent57 [1]. [8] shows that without PER, the learning score of a DQN agent may drop around 50%. However, PER requires even more frequent and irregular access to the ER memory, further exacerbating the memory-wall challenge.

In-memory computing, where computation is performed directly inside the memory array, is an effective computing paradigm for addressing the memory-wall challenge [10]. Associative memory (AM), a.k.a. content addressable memory (CAM), is an in-memory computing primitive that supports parallel search. AM can reduce the search time from $O(n)$ to $O(1)$ where n is the number of elements to be sought from. However, straightforward use of AM does not offer significant gains for PER since the basic tree-traversal steps for priority sampling are sparse and irregular. Hence, using hardware-software co-design, we design an AM based PER algorithm, AMPER and an AM based accelerator for AMPER. To the best of our knowledge, our proposed method is the first work that targets accelerating ER operations.

We specifically make the following contributions: (i) We investigate the DQN execution latency distribution under different ER memory and environment settings and identify that ER operations, especially the priority sampling process, are bottlenecks for implementing a low-latency DRL agent. (ii) We propose a novel AM-based prioritized experience replay (AMPER) algorithm with AM-friendly priority sampling operations, which replace the widely-used time-costly tree-traversal-based priority sampling in PER with TCAM searches, while preserving the learning performance. (iii) We propose two variants of AMPER, AMPER-k and AMPER-fr, using two

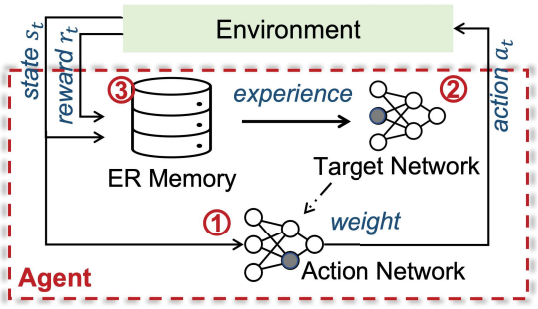

Figure 1: Illustration of a DQN agent interacting with the environment. The agent has three main components: (1) action network, (2) target network, and (3) ER memory.

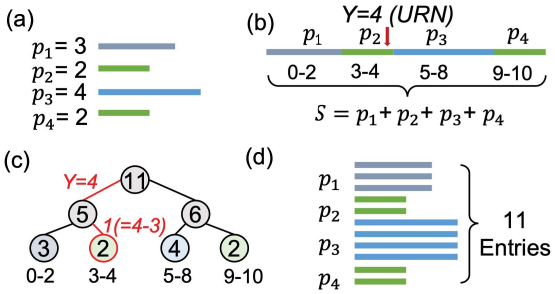

Figure 2: Illustration of PER implementation. (a) An example with 4 prioritized experiences. (b) The basic idea of sum-based sampling. (c) The sum-tree based implementation of (b). Leaf nodes contain the priority values. The search process of $Y = 4$ is highlighted in red. (d) A high-level conceptual view of AMPER for the example in (a).

AM-based nearest neighbor search operations: k-Nearest Neighbor and fixed-radius Nearest Neighbor, respectively, to trade off learning performance and latency. (iv) We design an AM-based in-memory computing hardware architecture by employing ternary CAMs (TCAMs) [9] to accelerate the AMPER algorithm. We devise a prefix-based query strategy to approximate fixed-radius Nearest Neighbor search with only a single low-latency TCAM search. (v) We evaluate AMPER on widely used OpenAI gym environments [3]. Our results show that AMPER achieves comparable learning performance as the PER algorithm. Our evaluations based on circuit-level simulations show that AMPER running on the AM-based in-memory computing hardware can achieve up to 270× latency improvement over PER running on GPU.

2 BACKGROUND AND MOTIVATION

Below we first present the basics of DQN and PER. We then briefly review the related work on DQN acceleration and AMs, especially TCAMs. Finally, we compare different existing ER techniques, and present the profiling data for a typical DQN implementing different ER techniques (uniform ER and PER) to further illustrate the latency performance characteristics.

2.1 DQN and Prioritized Experience Replay

DQN is a model-free, off-policy (i.e., using separate learning and action networks) DRL method which learns through past experiences [16]. Fig. 1 illustrates a typical DQN agent. At every time step t, the agent decides the action a_t via the action network, and uses that action to interact with the environment. The environment then transitions to a new state s_t and generates a reward r_t. At each time step, the state transition, the reward, and the action form an experience are stored in ER memory. A random batch of stored experiences are sampled at each time step and fed to the target network to train the agent. The agent learns from the state transitions and rewards by maximizing the global return, defined as the accumulated rewards from the start to the end.

Experience sampling plays an important role in the learning process. Prioritized experience replay (PER), as the state-of-the-art ER technique, frequently samples state transitions that lead to larger reward value change. PER has been empirically shown to improve the performance of the agent compared to the uniform ER [19] which samples the past distribution randomly following a uniform distribution. In PER, the priority sampling technique is deployed where each experience e_i is associated with a priority p_i determined by the relative magnitude of the temporal-difference error (TD-error). The probability that experience e_i is sampled is defined as $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ where $p_i > 0$. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. Also, PER needs to update the priority value of each sampled experience with a new TD-error after training is done.

Sampling experiences with PER in a large ERM can be very expensive. A sum-based method is widely used for the priority sampling and is adopted in PER. We illustrate the method in Fig. 2(b) using a simple example with 4 experiences as specified in Fig. 2(a). The sum of the four priorities is $S = p_1 + p_2 + p_3 + p_4 = 11$. A uniform random number (URN) Y is generated from the range $[0, S-1]$. Then, the sampled priority is the one corresponding to the region that Y falls into (e.g., $Y = 4$ falls in p_2 in Fig. 2(b) so the sampled priority is p_2). It is easy to see that the probability that Y falls into the region of p_2 is $P(2) = p_2/S$. Therefore, by using the sum-based representation, priority sampling is transformed into uniform sampling without knowing the data distribution. The sum-based method is typically realized with a data structure, sum tree, as shown in Fig. 2(c) where sampling is done by search on the sum tree structure. Also, the sum tree is updated when the priority value (leaf node) is updated. Thus, frequent updates and sampling operations in the DQN learning process incur many tree operations which require frequent access to memory and exhibit irregular memory access patterns, and cause longer latency.

2.2 Hardware Accelerators for DQN

Here we briefly review some representative previous work on DQN acceleration. An FPGA based accelerator is proposed in [21]. It focuses on accelerating the training and inference of the action and target network, and considers a small ER memory implementing the uniform sampling technique. Some other papers aim to accelerate distributed DQN, where multiple DQN agents work in a distributed fashion. For example, [22] proposes a customized network-on-chip design to solve the communication problems among the distributed agents. [15] exploits in-switch acceleration to reduce the network communication for gradient accumulation. Previous work usually assumes a small ER memory and ignores its acceleration. However, for the state-of-the-art DRL agents, a large ER memory is often needed and can incur long latency (more will be shown in Sec. 2.4).

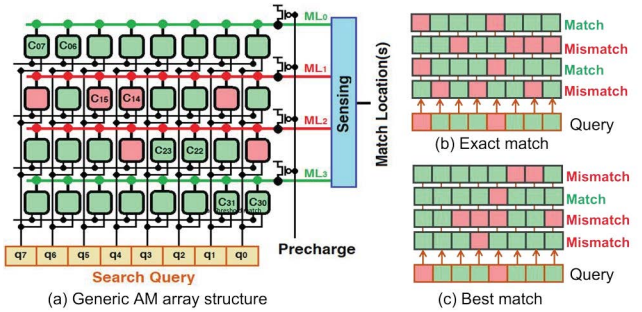

Figure 3: (a) Generic AM array structure (4×8 array) based on the NOR connection. Different match schemes: (b) exact match: the rows that are same as the input query; (c) best match: the row which has the shortest distance from input query is the best match. [9]

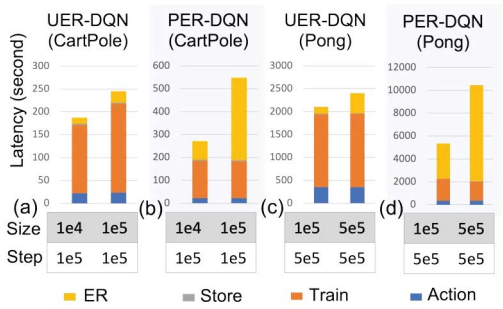

Figure 4: Latency breakdown for executing the UER-DQN and PER-DQN algorithm for the CartPole and Atari Pong environment. Size is the ER memory size and step is the total number of time steps.

2.3 In-memory computing and AM

Instead of moving data to the processing unit as in typical von Neumann machines, in-memory computing [20] performs computation directly inside the memory in order to solve the memory-wall [7] problem. Associative memories (AMs), also known as content addressable memory (CAMs), are in-memory-computing fabrics that support fast and energy efficient search. The two main operations of AMs are (1) search, where the address of the memory entry that matches the input query is identified and (2) write, where data entries are stored in the AM rows. AMs enable parallel searches of a given query against all data stored in memory in $O(1)$ time [9].

The most commonly used AM is a Ternary CAM (TCAM) where each element of queries and stored data can assume one of three states: 0, 1, and don't care ('x'). 'x' is a wildcard state which matches with both '0' and '1'. For a TCAM array with r rows and c columns (Fig. 3(a) [9]), all cells in a row are connected to a common matchline (ML) and each cell stores C_{ij}. During the search operation, each cell C_{ij} in row i performs an XNOR operation between its content and the query element q_j. If $C_{ij} = q_j$, C_{ij} matches the input query (denoted by green), and otherwise there is a mismatch (denoted by red). Each ML implements a logic OR operation of all the cells in the row to determine the result for that row.

Different sensing circuits can be designed to realize different match schemes. One typical match scheme is the exact match as shown in Fig. 3(b), which reports rows that "exactly match" the query for every single cell. Exact match search is the fastest search type due to its simple sensing requirement [9]. Another match scheme is best match, which reports the row with the least number of mis-matching cells. For best-match search, the discharge rate of the ML is proportional to the number of mis-match cells on the ML. Best match (Fig. 3(c)) search is widely used for nearest neighbor search [17]. To find the best match, it is possible to use analog-digital-converters to digitize the the ML voltage [12], which is a costly approach. Another approach is to use a winner-take-all circuit to find the row with the highest voltage (lowest discharge) [11]. This approach is more energy and area efficient than using analog-digital-converters but can be limited to finding best matches only within a certain number of mis-match cells. In this work, we will exploit the different match schemes to accelerate AMPER.

As discussed in Sec. 2.1, a tree-based method is employed to implement PER. Recent work [18] proposed to use AMs to accelerate a tree structure by mapping each path from the root node to a leaf node to one row of the AM. This kind of mapping can indeed accelerate the search process, but it exhibits poor latency performance for update since each update needs to write multiple rows in the AMs and is thus not desirable for implementing PER.

2.4 DQN Execution Latency Analysis

To accelerate DQNs, it is important to understand the latency distribution of different operations in a DQN. For the DQN agent shown in Fig. 1, at each time step, the following operations are done: store (storing a transition to the ER memory), ER operation (sampling a batch of transitions), train (training the target network), and action (action network inference to determine the action to take). Note that PER needs to update the priorities of the sampled transitions, which is also included in the latency of ER operation. We profile the DQN agent on the CartPole and Atari Pong environments running on a NVIDIA GTX 1080 GPU. Two kinds of ER are considered: uniform ER (UER) and PER. The network architectures are the same as in [16], i.e., a 3-layer MLP for the CartPole environment, and a 3-layer CNN for the Atari Pong environment. Usually, the ER memory size (the number of experiences) for a complex environment is set to 10^6 experiences. To study the relationship between the ER memory size and operation latency breakdown, we vary the size of ER memory. Furthermore, we consider two different total numbers of time steps.

Fig. 4 summarizes the profiling results with the corresponding ER memory size and the number of time steps. We observe several trends regarding the ER techniques and the ER memory size when comparing Fig. 4(a) with Fig. 4(b) and Fig. 4(c) with Fig. 4(d). First, ER operations in PER take much more time than in uniform ER. The reason is that despite the uniform random number generation process, sampling operation in PER (discussed in Sec. 2.1) needs to search on the sum tree structure and the tree needs to be updated with new priority values, which incur many tree-traversal steps. Second, a larger ER memory size can result in even longer time spent in ER over training due to the deeper tree depth. Third, when the ER memory size increases to 10^5, the ER operation takes nearly 50% of the total operation time. According to a study on the size of the ER memory [5], it is necessary to have a large ER memory to improve the learning performance (i.e., the global return) of the agent. Thus, in the state-of-the-art DQN, PER is a bottleneck in accelerating the learning process.

3 ASSOCIATIVE MEMORY BASED PER

This section presents our hardware-software co-design approach to accelerate PER. On the software side (Sec. 3.1–3.3), we introduce a novel algorithm AMPER which leverages AM-friendly priority sampling operations to approximate the original priority sampling technique. On the hardware side (Sec. 3.4), we design a AM-based in-memory computing architecture to support AMPER efficiently with fast search and update.

3.1 Overview of AMPER

As discussed in Sec. 2.4, PER faces memory access challenges due to frequent sampling and update operations. We aim to introduce an alternative PER method such that it can leverage the power offered by in-memory computing while preserving the learning performance offered by the original PER. In this subsection, we first present a high-level idea on approximating the priority sampling operation, and then give an overview of AMPER.

Intuitively, priority sampling aims to sample a higher-priority experience with a higher probability. Fig. 2(d) illustrates a straightforward way to transform priority sampling to uniform sampling. Here, we store multiple copies of the same priority, where the number of copies corresponds to the magnitude of the priority value. For example, we store three copies of p_1, two copies of p_2, etc., and a total of 11 entries. Now if we uniformly sample the 11 entries, the probability of sampling p_1 is p_1/S. It is easy to see that the sampling speed of this method should be much faster than the tree-based solution (Fig. 2(c)). However, the method would require a huge amount of memory, especially when the priority values are large.

Inspired by the idea shown in Fig. 2(d), we develop AMPER by using uniform sampling while minimizing memory requirements for storing priorities. Specifically, we propose to construct a subset of the priorities for uniform sampling such that the count of large priorities is higher than that of small priorities. The subset is referred to as the candidate set of priorities (CSP). Now if we uniformly sample the CSP, the larger priorities will be selected with higher probabilities. A key question then is how to construct the CSP so that the final learning performance would not be degraded.

To constructs CSP in AMPER we first divide all priorities into m groups, where m is a hyper parameter and bears some similarity to quantization level. Given the range of priority values as $[0, V_{max}]$, group g_i represents the value range $[\frac{V_{max}*i}{m}, \frac{V_{max}*(i+1)}{m}]$, where $g_0 \cup g_1 \cup \cdots \cup g_{m-1} = [0, V_{max}]$. For group g_i, the count of priorities in g_i is denoted by $C(g_i)$.

Consider the simplest case that we set m equal to the number of distinct priority values. Then, all the priorities within the same group have the same priority value. Fig. 5(a) depicts a representative distribution of priorities in a DQN, where each vertical bar corresponds to one group. Since the probability to sample the priorities in the same group should be equal, we can simply choose a subset of priorities from every group to form the CSP. If we let the size of the subset for g_i to be proportional to $C(g_i) \cdot V(g_i)$, where $V(g_i)$ denotes the priority value for group g_i, for a larger priority value $V(g_i)$, more priorities are included in the CSP.

The key idea behind AMPER is thus to approximate priority sampling with uniform sampling by constructing a representative CSP. However, several challenges still exist: (1) though the simplest method of setting m is straightforward and incurs little learning performance loss, the CSP can still be very large since the range of priority values is usually large. (2) Selecting priorities based on the value magnitude in run-time requires the priority list always sorted, which is costly to implement in CPU/GPU.

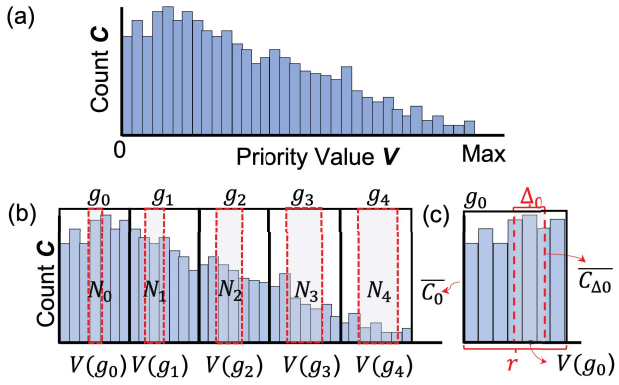

Figure 5: Key AMPER concepts: (a) Distribution of all priorities. X-axis is the priority value. Y-axis is the count corresponding to each distinct priority value. (b) Example of kNN based AMPER. 5 ($m = 5$) groups are used (separated by thick black lines), and the priorities in the red-dashed blocks are selected. (c) Example of frNN based AMPER. One group is shown as other groups follow the same idea.

3.2 Nearest Neighbor AMPER

In this section, we present a k-Nearest Neighbor(kNN) based priority sampling method to tackle the two challenges discussed above. This method would be able to exploit efficient search provided in AM. First, we set hyper parameter m much smaller than the number of distinct priority values to reduce the CSP size. In group g_i (for $0 \le i \le m-1$), the priority values are within the range of $[\frac{V_{max}*i}{m}, \frac{V_{max}*(i+1)}{m}]$ and the number of all the priorities with values within this range is $C(g_i)$. Following the idea discussed in Sec. 3.1, we need to determine the representative priority $V(g_i)$ and the subset size of g_i to construct CSP. For representative priority value $V(g_i)$, we randomly select a priority value in $[\frac{V_{max}*i}{m}, \frac{V_{max}*(i+1)}{m}]$, which preserves the randomness requirement in PER. We denote the subset size of g_i as N_i and set it as

$$N_i = \lambda \cdot V(g_i) \cdot C(g_i), i \in [0, m-1], \tag{1}$$

where λ is a scaling factor that scales the subset size and linearly correlates with the CSP size. λ is another hyper parameter which can be tuned to trade off learning performance and hardware cost. (The impacts of λ and m will be studied in Sec. 4.1).

Second, a kNN search process is employed to construct CSP with a few search steps without keeping the priority list sorted. Specifically, to obtain the subset of g_i, we choose N_i priorities with values closest to $V(g_i)$, which can leverage efficient search supported by AM. The rationale is that these priority values are good representatives for g_i. Note that simply randomly picking N_i values from g_i would be more expensive to implement in hardware and is thus avoided. The CSP is the union of the subset of each group as illustrated in Fig. 5(b). We uniformly sample from the resulting CSP to get the sampling result. Algorithm 1 (ignoring Line 9–12 for now) summarizes the AMPER method described above, which is referred to as **AMPER-k**. It can be seen that kNN search is the main operation in this AMPER implementation to find all N_i candidates.

3.3 Approximate Nearest Neighbor AMPER

As shown in Algorithm 1, the main operation in AMPER-k is kNN search. But two facts may limit the adoption of the method in an AM based architecture. First, the search function in AM requires a different sensing circuit than traditional AMs [9]. Typically, the sensing circuit for NN search incurs additional latency and area cost [13]. Also, several (k) search operations are needed to find all (k) neighbors. Second, to ensure we obtain N_i priorities for each group g_i, we need to keep track of the total priority count in each group, which requires additional circuitry. Below we introduce another variant of AMPER, which approximates kNN search with fixed-radius Nearest Neighbor search (frNN), AMPER-fr.

Fixed-radius nearest neighbor search, also known as C-Nearest Neighbor search, finds neighbors of the query within distance C. Fig. 5(c) illustrates the concept of AMPER-fr. The key idea is to employ parameter Δ_i representing the distance from value $V(g_i)$. Then, we use frNN search, find all the neighbors of $V(g_i)$ within distance Δ_i, and obtain a subset of g_i. Similar to AMPER-k, AMPER-fr constructs the CSP by taking the union of the subsets of all g_i's. Now, if we apply uniform sampling on the resulting CSP, we expect the sampled priority to be similar to that obtained by PER.

To ensure such a similarity holds, the key is determining an appropriate Δ_i. We derive the appropriate Δ_i according to Eqns. (2)-(4). Given distance Δ_i, the number of all priorities inside the range is $C_{\Delta i}$, which is expected to be an approximation of N_i in (1). Thus, the average number of all the distinct priority values within Δ_i is

$$\overline{C_{\Delta i}} = \frac{C_{\Delta i}}{\Delta_i} \approx \frac{N_i}{\Delta_i}, \tag{2}$$

where Δ_i can be determined by $\overline{C_{\Delta i}}$ and N_i. However, the challenge with this approach is that it is difficult to obtain the exact value of $\overline{C_{\Delta i}}$ because the distribution of the priorities changes from time to time as new experiences are put into ER memory. To address this issue, we propose to use the average number of all the distinct priority values within group g_i, $\overline{C_i}$, to approximate $\overline{C_{\Delta i}}$ as

$$\overline{C_{\Delta i}} \approx \overline{C_i} = C(g_i)/r, \tag{3}$$

where r is the group range size. From Eqn. (1), (2), and (3), we have

$$\Delta_i \approx \frac{N_i \cdot r}{C(g_i)} = \frac{\lambda \cdot V(g_i) \cdot C(g_i) \cdot r}{C(g_i)} = \lambda \cdot V(g_i) \cdot \frac{Vmax}{m} = \frac{\lambda'}{m} \cdot V(g_i). \tag{4}$$

Based on Eqn. 4, we can calculate Δi for each group to be used in the frNN search by only knowing $V(g_i)$, since λ' & k are hyper parameters. Thus, in AMPER-fr, for the i-th group, we search for neighbors of $V(g_i)$ within Δi distance (Fig. 5(c)) to construct the CSP. Algorithm 1 (ignoring lines 4–8) summarizes the sampling process in AMPER-fr. This design enables a faster AM search process and avoids the overhead of tracking priority counts in each group.

3.4 Hardware Support for AMPER

Here we present a hardware design to support AMPER. We elaborate on the high-level architecture and the operation flow, and provide details of the search methodology and additional circuits.

The AM-based AMPER accelerator architecture is shown in Fig. 6(a). The design supports parallel search with multiple TCAM arrays and contains a uniform random number generator (URNG), a query generator, and a candidate set buffer. The architecture

Algorithm 1: AMPER. (The `if` condition at lines 4–8 are for kNN variant and lines 9–12 describe the frNN variant.)

Input: All priorities p, group number m, scaling factors λ, λ', maximum priority value V_{max}, batch size b
Output: Sampled priority set in sp

```
// Construct the CSP.
1  CSP = [];
2  for i in range(m) do
3      V(g_i) = random.uniform(V_max/m · i, V_max/m · (i + 1)) ;
4      if kNN then
5          C(g_i) = count(p_i) in range(V_max/m · i, V_max/m · (i + 1));
6          N_i = round(λ · V(g_i) · C(g_i));
           // Add N_i neighbors of V(g_i) to CSP.
7          CSP.add(kNN(V(g_i), N_i));
8      end
9      else if frNN then
10         Δ_i = round(λ'/m · V(g_i));
           // Add V(g_i)'s neighbors in distance Δ_i.
11         CSP.add(frNN(V(g_i), Δ_i)) ;
12     end
13 end
   // Sample the CSP uniformly.
14 for j in range(b) do
15     id = random.uniform(len(CSP));
16     sp.add(CSP[id]);
17 end
18 return sp
```

works as follows: (1) The URNG generates a random search query $V(g_i)$ for each group (line 3 in Algorithm 1). (2) The query generator generates the corresponding search query for each group, and the query is sent to all TCAM arrays (lines 6&10 in Algorithm 1). (3) Multiple TCAM arrays work in parallel to find all matching entries, which are sent to the candidate set buffer (lines 7&11 in Algorithm 1). (4) In the last step, the URNG generates a batch of random numbers, and the corresponding entries in the candidate set buffer are accessed and used as the final output (lines 15&16 in Algorithm 1). For both the AMPER variants, the same dataflow can be deployed with minor differences in the query generator and the sensing circuit of the TCAM array, which are introduced below.

3.4.1 AMPER-k Search. The query generator for the kNN variant shown in Fig. 6(b1), implements Equ. 1 to calculate the expected CSP size N_i by using a Q-bit multiplier. The Q-bit input $V(g_i)$ will be output as the search query multiple times. The search time is controlled by N_i. As reviewed in Sec. 2.3, TCAM supports searching all the data stored in the TCAM array in one step. To realize the kNN search in AM, TCAM arrays with **best match sensing circuit** [4] can be deployed (red dotted block in Fig. 6(c)). For each search operation, the neighbor nearest to $V(g_i)$ is output, and multiple search operations are needed to find N_i nearest neighbors. Besides the multiple search steps needed, there are other challenges with the kNN implementation. Best match search requires more sophisticated sensing circuitry since an accurate comparison of the

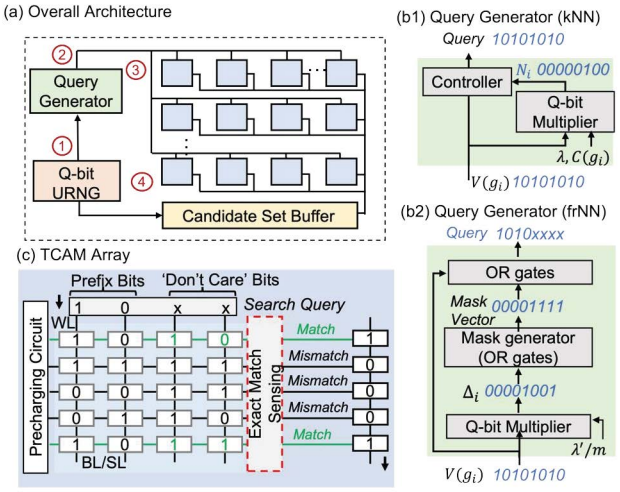

Figure 6: (a) Overview of the AM based architecture for supporting AMPER. Each blue-shaded box represents a TCAM array. (b1) Query generator design for AMPER-k. (b2) Query generator design for AMPER-fr with the prefix-based query generation. Query generation examples (Q=8 bits) are shown in (b1)(b2). (c) TCAM array with prefix-based query strategy and exact match sensing used for AMPER-fr.

number of matching cells is needed. Furthermore, the search accuracy can suffer significantly when the number of cells in a row is large and there are non-negligible device variations and noises [14].

3.4.2 AMPER-fr Search. For AMPER-fr search, we devise a **prefix-based query strategy**, an efficient query mapping technique, which approximates the search radius by using the bit properties of fixed-point values. This approach only requires **exact-match** TCAMs which employ very simple sensing circuitry since only match or mismatch need to be differentiated. Furthermore, only one search operation is needed to get all candidates. The prefix-based query strategy follows the steps below to select all neighbors within Δ_i distance of $V(g_i)$.

First, the query generator is designed as shown in Fig. 6(b2), which consists of a Q-bit multiplier, a mask generator, and Q OR gates. A three-step prefix generation works as follows: 1) The multiplier generates search range Δ_i following Equ. 4. 2) The mask generator finds the position of the leftmost '1' in Δ_i, called 'p', which determines the position of prefix bits and don't care bits in the mask vector. In the mask vector, all bits to the left of 'p' are set to '0' and all bits to the right of 'p' (including 'p') are set to '1'. The mask generator is implemented using OR gates. 3) Given the input V_{gi} and mask vector of Δ_i, the OR gates generate a query composed of prefix bits and don't care bits. An 8-bit (Q=8) prefix query generation example is shown in Fig. 6(b2) where 'p' is 4. By employing the TCAMs, all the rows that match the query are identified. For the example in Fig. 6(c), query 10xx will match with the entries within the range (1000,1011). Thus, the number of don't care bits in the query corresponds with the size of the search range. Note that this prefix-based mapping does introduce some approximation error when $V(g_i)$ and Δ_i are not powers of 2 since the accepted range can only be powers of 2. Detailed latency comparison between the two variants' implementation will be presented in Sec. 4.2.

3.4.3 Update in AMPER. As we mentioned in Sec. 2.1, the sum tree in original PER implementation is updated when updating the priority value (leaf node), which also incurs lots of tree-traversal steps. However, the priority update operation in the proposed AMPER is relatively simple because each priority has only one copy in the ER memory. To update the priority, we write the new priority value in AM directly using the write port of TCAM arrays, which is also much faster than the original PER.

4 EVALUATION

We evaluate the proposed AMPER design with respect to algorithm-level performance and execution latency. We first present the algorithm-level performance study in Sec. 4.1. Then array-level and end-to-end latency study is discussed in Sec. 4.2.

4.1 Algorithm-level Performance Study

4.1.1 Sampling Error Study. Since AMPER adopts a novel AM-friendly priority sampling concept, it is important to compare AMPER and PER regarding the sampling performance. Specifically, we compare the sampling results from PER and AMPER. First, we generate a random data list with size 10000 from an uniform distribution within the range [0,1] and sample it with PER and AMPER, respectively. The sampling is repeated with batch size 64 for 100 runs. The sampling results distribution is visualized in Fig. 7(a), where two variants of AMPER both generate similar results as the standard PER with the curve mostly overlapped.

To further analyze the sampling difference between AMPER and PER, we quantify the difference using the metric Kullback-Leibler (KL) Divergence, a measure of how one probability distribution is different from another reference probability distribution with the unit 'nat'. Smaller KL Divergence values indicate more similar distributions. Given two discrete distributions P, Q, the KL divergence is defined as $KL(P,Q) := SUM(P[i] * log(P[i]/Q[i]), i)$.

As we discussed in Sec. 3.2, the scaling factor λ and group number m are two key hyper parameters in CSP construction. We vary the values of λ and m and repeat the sampling process with batch size 64 for 100 runs to generate different sampling results. Fig. 7(b)(c) shows the KL divergence values between the AMPER and the PER sampling results under different hyper parameter combinations. Group number m is shown along the y-axis for the two figures, increasing from 2 to 12. X-axis depicts the scaling factor λ/λ' which linearly correlates with the size of the CSP. As shown in Fig. 7(b) and (c), for both AMPER variants, increasing group number m and scaling factor λ/λ' decreases the KL Divergence value, which means less sampling error. AMPER introduces a large sampling error when the group number and scaling factor are very small (upper left corner), say $m = 2, \lambda = 0.05$. However, at the bottom right corner that AMPER has less than 300 nats KL divergence, which is quite similar to the original PER. For reference, the KL Divergence value between uniform sampling and PER sampling is around 9000 nats, and the KL Divergence value between different runs of PER is around 140 nats. Thus, by choosing proper hyper parameters, AMPER can achieve a similar sampling performance as PER. Comparing Fig. 7(b) and (c), AMPER-fr also achieves comparable performance as AMPER-k. Later in Sec. 4.2, we will further discuss the impact of hyper parameters on execution latency.

Fig. 7(d) studies the sampling error under different ER memory size. We vary the ER memory size from 5000 to 20000 for AMPER-k. For each ER memory size, the group number is set to 4/8/12, and

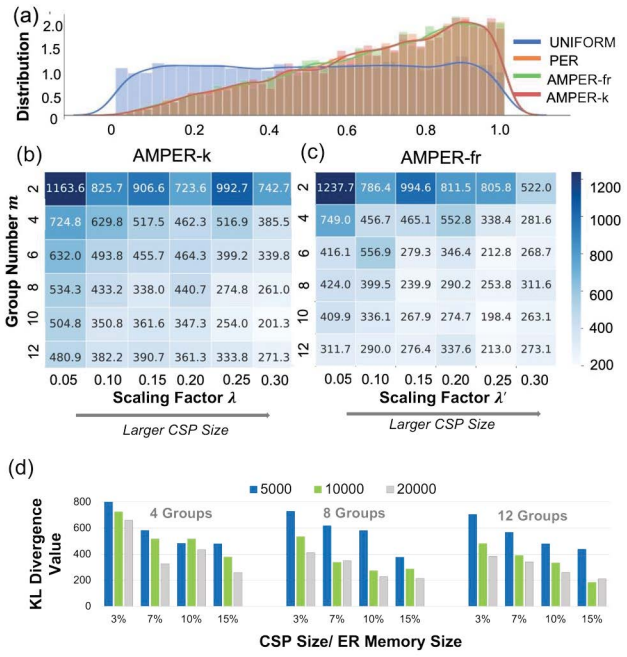

Figure 7: Sampling error study. (a)Visualization of the sampling results from Uniform, AMPER-k, AMPER-fr and PER methods. KL Divergence study of (b) AMPER-k (c) AMPER-fr under different group numbers and scaling factors. (d) KL Divergence study of AMPER-k for different ER memory sizes (5000, 10000, 20000).

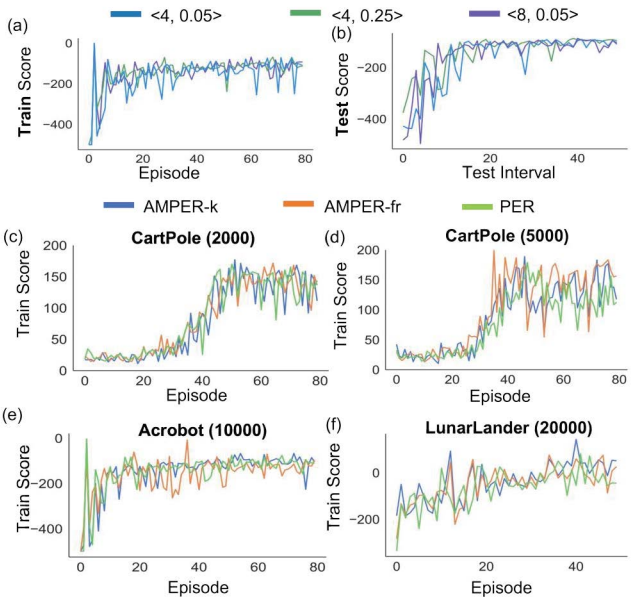

Figure 8: (a) train scores and (b) test scores when training with different hyper parameters for AMPER-k on the Acrobot environment. Group size and scaling factor are set to <4, 0.05>, <4, 0.25>, <8, 0.05>, respectively. Training scores of the DQN agent using PER, AMPER-k, and AMPER-fr with different ER memory sizes: (c) CartPole with size 2000; (d) CartPole with size 5000; (e) Acrobot with size 10000; (e) LunarLander with size 20000. The scores are averaged over 3 runs.

the x-axis is CSP ratio (CSP size / ER memory size). Fig. 7(d) shows that the findings in Fig. 7(b)(c) still hold for different ER memory sizes. Also, under the same m and the CSP ratio, AMPER achieves better sampling performance when the ER memory size becomes larger. The same trends hold for AMPER-fr.

4.1.2 DQN Learning Performance Study. To study the performance of AMPER in DQN learning, we implemented AMPER using PyTorch and tested it on the learning environments CartPole, Acrobot, and LunarLander provided by OpenAI Gym [3]. The action/target networks and their basic hyper parameters are set as [8]. We fix the number of steps for each environment. If the ER memory is full, it discards the oldest experience. The training score is the return of each training episode, and the test score is the average return of 10 episodes. The return is defined as the accumulated reward over an episode. Each environment defines its own rewards. For Cartpole, +1 reward is given at a timestep if the pole remains upright. The Acrobot environment gives a reward of -1 at each time step before the problem is solved. For LunarLander, the environment gives either positive or negative rewards at each step. The higher the score, the better the agent learning performance.

We first evaluate the relationship between the sampling error and DQN learning performance. We choose three sets of $<m, \lambda>$ combination: <4, 0.05> / <4, 0.25> / <8, 0.05>, which correspond to KL divergence value of 724.8 / 516.9 /534.3 nats, respectively. Fig. 8(a)(b) show the training score and test score curves of the DQN agent on the Acrobot environment with ER memory size 10000. It can be seen that the <4, 0.05> (blue curve) combination, which has the largest sampling error, learns slowest and has the most unstable training curve compared with the other two that exhibit similar learning performance. However, the three settings still reaches to similar final score (Fig. 8(b)) and AMPER works well in DQN learning even with large sampling error (around 700 nats).

Fig. 8(c)-(e) show the learning curves of the DQN agent using PER and the proposed AMPER on the three environments. AMPER-fr and AMPER-k exhibit comparable learning speed and final score as PER. Table 1 summarizes the test scores for the different tasks and ERM sizes. AMPER achieves little score degradation compared with PER. Moreover, AMPER-k achieves even better performance in some cases (CartPole-2000, Acrobot, LunarLander).

Comparing AMPER-k and AMPER-fr, the kNN variant has a more stable learning process and better final score than the frNN variant. The reason is that Eqn. (3) incurs approximation errors in the frNN implementation. If the average count of the selected subset (i.e., $\overline{C_{\Delta i}}$) is quite different from the average group count ($\overline{C_i}$), an error is introduced when calculating Δ_i. However, AMPER-fr still exhibits similar performance as PER in most cases, and provides much faster speed to be shown in Sec. 4.2.

4.2 Hardware Performance Study

4.2.1 Experimental Setup. To evaluate the proposed hardware accelerator for AMPER, we developed RTL-level Verilog models for URNG circuits and the query generator (QG) and synthesized them with Cadence Encounter for a CMOS 45nm library[1]. Each priority entry is represented with INT-32 bits. The bit-width Q is set

[1]We used 45nm CMOS technology library in order to be consistent with the TCAM data reported in [17].

Table 1: Test score comparison of PER, AMPER-k and AMPER-fr on the OpenAI environments (CartPole, Acrobot, LunarLander).

Env	Size	PER	AMPER-k	AMPER-fr
CartPole	2000	162.20	**180.13**	154.18
CartPole	5000	**177.32**	173.20	173.25
Acrobot	10000	-89.39	**-88.89**	-93.69
LunarLander	20000	185.33	**200.10**	161.50

Table 2: Latency of AMPER hardware components.

Component	TCAM Array (Exact [17])	TCAM Array (Best [4])	CSB (0.03MB)
Operation	Search/Write	Search/Write	Read / Write
Delay (ns)	0.58 / 2.0	1.0 / 2.0	0.78 / 0.78
Component	URNG	QG (kNN)	QG (frNN)
Delay (ns)	1.71	3.57	2.02

to 32 for each component. The URNG is implemented with the 32-bit linear feedback shift register. The latency consumed by the candidate set buffer are calculated using CACTI [2]. A candidate set buffer (CSB) with a size of 0.3MB is used, which can hold 8000 entries in total. We employ the CMOS-based 16T TCAM design with the best match [4] and exact match sensing circuits [17] for the proposed hardware accelerator. Each TCAM array is 64 rows × 64 columns, where each row stores a priority entry. Multiple arrays (e.g. 128 arrays for ER memory size 8,192) are needed to store all the priorities. Table. 2 summarizes the latency of each component. The latency of PER is measured on the system with Intel i5-8600k CPU and Nvidia RTX 1080 GPU. Sampling is done in batches of 64, that is, 64 priorities are returned after each sampling.

4.2.2 Latency Evaluations. We first compare the performance of the proposed accelerator with the GPU implementation. The latency is measured for per batch sampling. To ensure the best learning performance, we set m to 20 and the CSP ratio to 15%. Fig. 9(a) summarizes the comparison for ER memory sizes 5000, 10000, and 20000. AMPER-k and AMPER-fr are 55×–170× and 118×–270× faster than the GPU implementation, respectively. Note that AMPER runs slower on GPU/CPU than the original PER because the nearest neighbor search operation is time-consuming on GPU/CPU. However, our hardware-software co-design approach achieves significant latency improvements over PER. Moreover, based on the data shown in Fig. 9(a), AMPER-frNN achieves ~2× latency improvement compared to AMPER-k. Although the parallel TCAM search ability is exploited in both variants, AMPER-fr achieves better performance due to the simple sensing circuit design. According to the data in Table 2, for each search operation, the TCAM array with best match sensing incurs 1.7× latency compared with the exact match one due to the more complicated sensing circuit. Furthermore, the number of search operations is reduced by using frNN search compared with kNN search.

Fig. 9(b) investigates the end-to-end latency of AMPER-fr and AMPER-k with different group numbers, where the CSP size ratio is fixed to 0.15. The ER memory size is set to 10000 for both implementations. For both implementations, increasing group number has a small impact on the latency. This is because the TCAM array search is done in parallel, which is much faster than other components

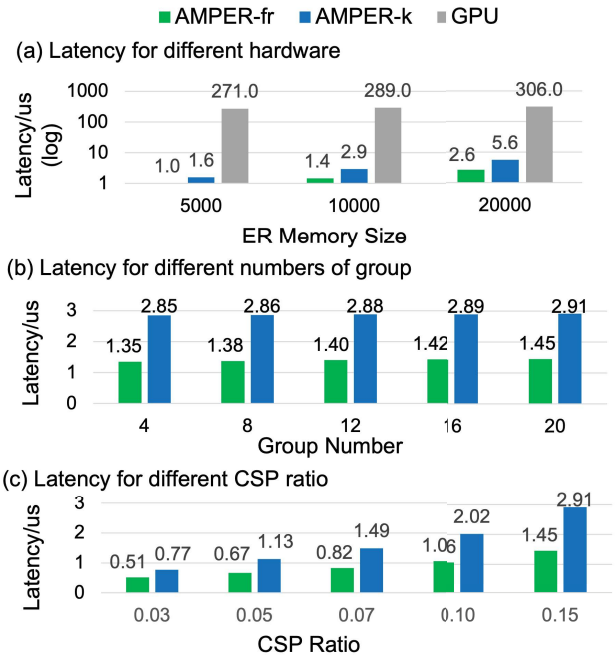

Figure 9: End-to-end latency for AMPER-fr and AMPER-k. (a) Comparison with GPU implementation. (b) With CSP ratio at 0.15. Varying group number m from 4 to 20. (c) With group number m at 20. Varying CSP ratio from 0.03 to 0.15.

(Table 2), especially the candidate set buffer write operations. Thus the additional search operations introduced by increasing group number has little impact on the end-to-end latency.

The end-to-end latency for AMPER-fr and AMPER-k for different CSP sizes is studied in Fig. 9(c), where the group number is fixed to 20. Fig. 9(c) shows that the latency of both AMPER-fr and AMPER-k increases linearly with the CSP size as the latency is now dominated by the candidate set buffer throughput. As discussed in Fig. 7, increasing both the group number and the CSP size helps improve the algorithm-level performance of the two variants. However, according to the data in Fig. 9(b)(c), increasing the group number is a better option as it incurs limited additional latency to get better sampling performance.

5 CONCLUSION

In this paper, we propose a hardware-software codesign approach AMPER to accelerate PER in the state-of-the-art DRL agent. AMPER employs the AM-based search operation to approximate PER. An in-memory-computing hardware architecture based on AM is designed to support AMPER. AMPER shows comparable learning performance as the PER and achieves 55×~270× latency improvement compared with PER implemented on GPU.

ACKNOWLEDGMENTS

This work was supported in part by NSF CCF-2028879 and CCF-1640081, and by ASCENT, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. 2020. Agent57: Outperforming the atari human benchmark. In *International Conference on Machine Learning*. PMLR, 507–517.

[2] Rajeev Balasubramonian and et.al. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *TACO* 14, 2 (2017), 1–25.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540

[4] S. Dutta, A. Khanna, H. Ye, M.M. Sharifi, A. Kazemi, M.San Jose, K.A. Aabrar, J.G. Mir, M. Niemer, X.S. Hu, and S. Datta. 2021. Lifelong Learning with Monolithic 3D Ferroelectric Ternary Content-Addressable Memory. In *2021 IEEE International Electron Devices Meeting (IEDM)*. 1–4. https://doi.org/10.1109/IEDM19574.2021.9720495

[5] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. 2020. Revisiting fundamentals of experience replay. In *International Conference on Machine Learning*. PMLR, 3061–3071.

[6] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. 2019. Learning to Walk Via Deep Reinforcement Learning.. In *Robotics: Science and Systems*.

[7] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[8] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.

[9] Xiaobo Sharon Hu, Michael Niemier, Arman Kazemi, Ann Franchesca Laguna, Kai Ni, Ramin Rajaei, Mohammad Mehdi Sharifi, and Xunzhao Yin. 2021. In-memory computing with associative memories: a cross-layer perspective. In *2021 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 25–2.

[10] Daniele Ielmini and H-S Philip Wong. 2018. In-memory computing with resistive switching devices. *Nature electronics* 1, 6 (2018), 333–343.

[11] Mohsen Imani, Xunzhao Yin, John Messerly, Saransh Gupta, Michael Niemier, Xiaobo Sharon Hu, and Tajana Rosing. 2019. Searchd: A memory-centric hyperdimensional computing with stochastic training. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2422–2433.

[12] Geethan Karunaratne, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. 2020. In-memory hyperdimensional computing. *Nature Electronics* 3, 6 (2020), 327–337.

[13] Arman Kazemi, Mohammad Mehdi Sharifi, Ann Franchesca Laguna, Franz Müller, Ramin Rajaei, Ricardo Olivo, Thomas Kämpfe, Michael Niemier, and X Sharon Hu. 2021. In-memory nearest neighbor search with fefet multi-bit content-addressable memories. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1084–1089.

[14] Arman Kazemi, Mohammad Mehdi Sharifi, Ann Franchesca Balon Laguna, Franz Muller, Xunzhao Yin, Thomas Kampfe, Michael Niemier, and X Sharon Hu. 2021. FeFET Multi-Bit Content-Addressable Memories for In-Memory Nearest Neighbor Search. *IEEE Trans. Comput.* (2021).

[15] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 279–291.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[17] Kai Ni, Xunzhao Yin, Ann Franchesca Laguna, Siddharth Joshi, Stefan Dünkel, Martin Trentzsch, Johannes Müeller, Sven Beyer, Michael Niemier, Xiaobo Sharon Hu, et al. 2019. Ferroelectric ternary content-addressable memory for one-shot learning. *Nature Electronics* 2, 11 (2019), 521–529.

[18] Giacomo Pedretti, Catherine E Graves, Sergey Serebryakov, Ruibin Mao, Xia Sheng, Martin Foltin, Can Li, and John Paul Strachan. 2021. Tree-based machine learning performed in-memory with memristive analog CAM. *Nature communications* 12, 1 (2021), 1–10.

[19] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint:1511.05952* (2015).

[20] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. 2020. Memory devices and applications for in-memory computing. *Nature nanotechnology* 15, 7 (2020), 529–544.

[21] Jiang Su, Jianxiong Liu, David B Thomas, and Peter YK Cheung. 2017. Neural network based reinforcement learning acceleration on fpga platforms. *ACM SIGARCH Computer Architecture News* 44, 4 (2017), 68–73.

[22] Ying Wang, Mengdi Wang, Bing Li, Huawei Li, and Xiaowei Li. 2020. A many-core accelerator design for on-chip deep reinforcement learning. In *ICCAD*. 1–7.