Associative Memory Based Experience Replay for Deep Reinforcement Learning

Mengyuan Li, Arman Kazemi, Ann Franchesca Laguna and X. Sharon Hu

Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, Indiana, USA

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 55x to 270x 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 [1]. 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 [2], 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) [2].

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 [3] 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) [4], deploying priority sampling technique, is widely used in the state-of-the-art DQN implementations like Rainbow [5] and Agent57 [6]. [5] 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 [7]. 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 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) [8] to accelerate the AMPER algorithm. We devise
We evaluate AMPER on widely used OpenAI gym environments [9].

Finally, we compare different existing ER techniques, and

defined as $e_{\text{error}}$ (TD-error). The probability that experience

determined by the relative magnitude of the temporal-difference

deployed where each experience $e$ is sampled from.

improve the performance of the agent compared to the uniform

larger reward value change. PER has been empirically shown to

process. Prioritized experience replay (PER), as the state-of-the-

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 experi-

ences [2]. 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 transi-

tions 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 [4] 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 $P(i) = \frac{p_i^\alpha}{\sum_k p_k}$ where $p_i > 0$. The exponent $\alpha$ 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 [10]. 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, [11] proposes a customized network-on-chip

design to solve the communication problems among the distributed

agents. [12] 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).
2.3 In-memory computing and AM

Instead of moving data to the processing unit as in typical von Neumann machines, in-memory computing [13] performs computation directly inside the memory in order to solve the memory-wall [3] 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 [8].

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) [8]), 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 [8]. Another match scheme is best match, which reports the row with the least number of mismatching 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 [14]. To find the best match, it is possible to use analog-digital-converters to digitize the ML voltage [15], 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) [16]. 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 [17] 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 [2], 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 [18], 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 DQN 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 construct 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 $[0, V_{\text{max}}]$, group $g_i$ represents the value range $[\frac{V_{\text{max}}\cdot i}{m}, \frac{V_{\text{max}}\cdot (i+1)}{m}]$, where $g_0 \cup g_1 \cup \cdots \cup g_{m-1} = [0, V_{\text{max}}]$. 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 $g_i$ 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.

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 $0 \leq i \leq m - 1$), the priority values are within the range of $[\frac{V_{\text{max}}\cdot i}{m}, \frac{V_{\text{max}}\cdot (i+1)}{m}]$ 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 $[\frac{V_{\text{max}}\cdot i}{m}, \frac{V_{\text{max}}\cdot (i+1)}{m}]$, which preserves the randomness requirement in PER. We denote the subset size of $g_i$ as $N_i$ and set it as

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

where $\lambda$ is a scaling factor that scales the subset size and linearly correlates with the CSP size. $\lambda$ is another hyper parameter which can be tuned to trade off learning performance and hardware cost. (The impacts of $\lambda$ 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 [8]. Typically, the sensing circuit for NN search incurs additional latency and area cost [19]. 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 $\Delta_i$ representing the distance from value $V(g_i)$. Then, we use frNN search, find all the neighbors of $V(g_i)$ within distance $\Delta_i$, 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 $\Delta_i$. We derive the appropriate $\Delta_i$ according to Eqs. (2)-(4). Given distance $\Delta_i$, 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 $\Delta_i$ is

$$C_{\Delta_i} = \frac{C_{\Delta_i}}{\Delta_i} \approx \frac{N_i}{\Delta_i}$$

(2)

where $\Delta_i$ can be determined by $C_{\Delta_i}$ and $N_i$. However, the challenge with this approach is that it is difficult to obtain the exact value of $C_{\Delta_i}$ 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 $g_i$, $C_{\Delta_i}$, to approximate $C_{\Delta_i}$ as

$$C_{\Delta_i} \approx C_{\Delta_i} = C(g_i)/r.$$

(3)

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

$$\Delta_i \approx \frac{N_i \cdot r}{C(g_i)} = \frac{\lambda \cdot V(g_i) \cdot C(g_i)}{C(g_i)} \cdot r = \lambda \cdot V(g_i) \cdot \frac{V_{\max}}{m} = \frac{\lambda}{m} \cdot V(g_i).$$

(4)

Based on Eqn. 4, we can calculate $\Delta_i$ for each group to be used in the frNN search by only knowing $V(g_i)$, since $\lambda$ & $k$ are hyper parameters. Thus, in AMPER-fr, for the $i$-th group, we search for neighbors of $V(g_i)$ within $\Delta_i$ 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 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(b), implements Eqn. 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 [20] 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 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 [21].

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 $\Delta_i$ 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 $\Delta_i$ following Eqn. 4. 2) The mask generator finds the position of the leftmost ‘1’ in $\Delta_i$, named ‘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 $V(g_i)$ and mask vector of $\Delta_i$, 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 $\Delta_i$ 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. For two discrete distributions $P, Q$, the KL divergence is defined as $KL(P, Q) = SUM(P[i] \cdot log(P[i]/Q[i]), i)$.

As we discussed in Sec. 3.2, the scaling factor $\lambda$ and group number $m$ are two key hyper parameters in CSP construction. We vary the values of $\lambda$ 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 $\lambda$ 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 $\lambda$ 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,
We fix the number of steps for each environment. If the ER memory which is quite similar to the original PER. For reference, the 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).

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 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 [9]. The action/target networks and their basic hyper parameters are set as [5]. 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, λ> 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., $\overline{C_{\lambda i}}$) is quite different from the average group count ($\overline{C_{\lambda i}}$), an error is introduced when calculating $\Delta_i$. 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. Each priority entry is represented with INT-32 bits. The bit-width Q is set 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 [22]. 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 [20] and exact match sensing circuits [14] 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 performance, we used 45nm CMOS technology library in order to be consistent with the TCAM data reported in [14].
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 55x–170x and 118x–270x 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 (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.

REFERENCES

[1] T. Haarnoja, et al. Learning to walk via deep reinforcement learning. In Robotics: Science and Systems, 2019.
[2] V. Mnih, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.
[3] S. Han, et al. Eie: Efficient inference engine on compressed deep neural network. ACM SIGARCH Computer Architecture News, 44(3):243–254, 2016.
[4] T. Schaul, et al. Prioritized experience replay. arXiv preprint:1511.05952, 2015.
[5] M. Hessel, et al. Rainbow: Combining improvements in deep reinforcement learning. In Thirty-second AAAI conference on artificial intelligence, 2018.
[6] A. P. Badia, et al. Agent57: Outperforming the atari human benchmark. In International Conference on Machine Learning, pages 507–517. PMLR, 2020.
[7] D. Ielmini et al. In-memory computing with resistive switching devices. Nature electronics, 1(6):333–343, 2018.
[8] X. S. Hu, et al. In-memory computing with associative memories: a cross-layer perspective. In 2021 IEEE International Electron Devices Meeting (IEDM), pages 25–2. IEEE, 2021.
[9] G. Brockman, et al. Openai gym, 2016.
[10] J. Su, et al. Neural network based reinforcement learning acceleration on fpga platforms. ACM SIGARCH Computer Architecture News, 44(4):68–73, 2017.
[11] Y. Wang, et al. A many-core accelerator design for on-chip deep reinforcement learning. In ICCAD, pages 1–7, 2020.
[12] Y. Li, et al. Accelerating distributed reinforcement learning with in-switch computing. In 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), pages 279–291. IEEE, 2019.
[13] A. Sebastian, et al. Memory devices and applications for in-memory computing. Nature nanotechnology, 15(7):529–544, 2020.
[14] K. Ni, et al. Ferroelectric ternary content-addressable memory for one-shot learning. Nature Electronics, 2(11):521–529, 2019.
[15] G. Karunaratne, et al. In-memory hyperdimensional computing. Nature Electronics, 3(6):327–337, 2020.
[16] M. Imani, et al. Searchd: A memory-centric hyperdimensional computing with stochastic training. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(10):2422–2433, 2019.
[17] G. Pedretti, et al. Tree-based machine learning performed in-memory with memristive analog cam. Nature communications, 12(1):1–10, 2021.
[18] W. Fedus, et al. Revisiting fundamentals of experience replay. In International Conference on Machine Learning, pages 3061–3071. PMLR, 2020.
[19] A. Kazemi, et al. In-memory nearest neighbor search with fefet multi-bit content-addressable memories. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1084–1089. IEEE, 2021.
[20] S. Dutta, et al. Lifelong learning with monolithic 3d ferroelectric ternary content-addressable memory. In 2021 IEEE International Electron Devices Meeting (IEDM), pages 1–4, 2021.
[21] A. Kazemi, et al. Fefet multi-bit content-addressable memories for in-memory nearest neighbor search. IEEE Transactions on Computers, 2021.
[22] R. Balasubramonian et al. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. TACO, 14(2):1–25, 2017.